RAPID PROTOTYPING OF DISTRIBUTED SYSTEMS OF ELECTRONIC CONTROL UNITS IN VEHICLES

James Brady

A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy, Loughborough University

June 2019

Abstract

Existing vehicle electronics design is largely divided by feature, with integration taking place at a late stage. This leads to a number of drawbacks, including longer development time and increased cost, both of which this research overcomes by considering the system as a whole and, in particular, generating an executable model to permit testing. To generate such a model, a number of inputs needed to be made available. These include a structural description of the vehicle electronics, functional descriptions of both the electronic control units and the communications buses, the application code that implements the feature and software patterns to implement the low-level interfaces to sensors and actuators.

Executable models generated by this approach are subject to physical and electronic constraints. A number of alternative solutions can be generated by this approach and the merits of these are considered here. The optimization of the design space, that is the reduction of the total number of candidate solutions considered so as to identify the feasible solutions before execution of the design tools and exclude wasteful searches of infeasible architectures and software/hardware configurations, is one novel outcome of this research.

This research provides an end to end example of the design of a distributed system of vehicle electronics by the development of software design tools and the integration of software algorithms culminating in a deployment of distributed vehicle control software on shared electronic control units (ECUs) to reduce cost, mass and wiring.

The approach of this research is to obtain requirements for a system of vehicle features and to model them in software so as to enable the design of the applications software and algorithms to operate the features before using further software models to allocate the features to clusters of shared ECUs that can communicate with other clusters that were designed in isolation.

Outcomes of this research include a method of designing a complete distributed system of ECUs from systems diagrams that offer the possibility of generation of source code as well as a tool for the optimisation of software/hardware allocation on the ECUs. Although this research focuses on an automotive road vehicle application, other industries can benefit from its output. Allocation of ECUs to software features arises anywhere that a distributed system of microcontrollers can be implemented, for example in transport generally including railway trains, aeroplanes, ships and submersibles or in medical patient healthcare with the design of modularised intensive care monitoring equipment and surgical robotics. Whilst this research concentrates on wired devices and systems, wireless applications would allow for the communication between autonomous entities (robots, drones, satellites, self-driving cars for example) that move around in space and either communicate so as to work together in a hive mentality for greater efficiency/power or to avoid collision by updating information regarding their relative positions and proximity to act upon these and maintain separation in a controlled fashion. The possibilities are endless and are limited only by imagination or the lack of it.

Acknowledgements

I would like to thank my academic supervisors, Professor Charles Dickerson and Dr David Mulvaney for believing in me at interview in 2014 and agreeing to give me this opportunity to challenge myself beyond what I might have considered possible. I acknowledge my industrial supervisor, Timothy Snell of Jaguar Land Rover (JLR), for his time that he set aside for meetings at JLR's Gaydon premises and, to JLR, thank you for sponsoring my studentship and granting me access to the 'Program for Simulation Innovation' (PSi) Project.

To my family, sister Clare (no, she's not a nun), brother Tim (not a monk) and brother Paul, I thank you for being there to support me through difficult times and to indulge me when I was in the mood to share the highs.

To my late parents, Rose, you told me that if I thought about things too much I would go mad. It nearly happened once or twice over the last four years and Pat, I'm sorry that this was always going to be what I did with my life and not playing for Everton FC.

To Janice, for staying awake that one time when I was talking about mathematics and for surprising me by demonstrating just how much of what I've learned has rubbed off on you, I need you to know I've always loved you and always will.

To Sue Smith, my friend who turned up again after 18 years, thank you for the hugs without which I wouldn't have survived 2018.

To everyone at CREST, I can't name you individually because I will either reveal my favouritism or forget one of you who have all meant so much to me. I am truly grateful to every one of you that has ever played table tennis with me, stood in the kitchen and listened to my deeply disturbing (is that disturbed?) questions, invited me to a party or brought birthday treats, welcomed me into your circle for Prosecco on the mezzanine or just said hello whenever I've walked into the office.

To Jordan Dawson, Andrew Rowe, Natalie Tebbett and all the members of the PhD SSN, you provided a service that gave me something to look forward to every Tuesday that wasn't always about the task of completing a PhD.

To the staff at Graduate House, your smiling faces every Tuesday lifted me and gave me the strength to carry on when sometimes giving in felt like an easier option. Duncan Stanley, for helping me to deal with the disappointment of the feedback on my first year progression report and giving me the advice and guidance to turn things around, I will always be truly grateful. Zoe Critchlow, I don't know how many times I've nominated you for an 'LSU Post Graduate Award' but you will win one someday. You have to, because you are awesome.

To everyone in IT Services, especially Ray Chung, David Brock and Stuart Clow, thank you for your continued and unwavering support from day one.

Table of Contents

List of Figures

List of Tables

1. Introduction

This research sets out to answer many interrelated questions regarding the design of electronic systems in road vehicles. Time to market, overall cost, number of ECUs needed, automation of source code for control algorithms and the effect of overall vehicle mass on performance are all evaluated to find savings that could boost profit and performance.

1.1 Rationale

Benefits are reaped from the reduction in the time taken to design the hardware/software architecture due to the follow on reduction in time to market for the completed vehicle. Reduction in the number of ECUs needed to support a given number of features brings down the cost of the vehicle and improves performance by reducing the overall mass so as to better overcome inertia during acceleration events such as starting, stopping and changing direction.

Acquisition of a suitable heuristic algorithm to allocate software to the optimal number of ECUs was sought so that the allocation process could be automated and the search space reduced to speed up the process. The research aimed to reduce the execution times for heuristic methods, which searched a subset of the solution space by eliminating non-feasible solutions. A trade-off between allowing duplicate DNA and preventing the same DNA from being evaluated more than once directed the research towards seeking a model that encouraged jumping from a local optimum so as to allow the potential for finding a global optimum rather than converging either too quickly or on a less useful solution. Measures of success in reducing execution times are evidenced in the number of iterations required to test all DNA and elapsed time in seconds for the program to execute. Whilst the latter is machine dependent, the first is not.

Time that could be saved by the partial automation of the creation of executable code on ECUs was an intended achievement of this research – xtUML diagrams supported the generation of some source code although the limited success of this is discussed in later chapters.

To support the assertions of the research from the outset, that a system could be designed starting from requirements and a diagram of the model, an end to end model from system requirements to the implementation of an executable model verified and validated with appropriate microcontrollers and communications protocols and HILs was an expected achievement and an outcome of the research.

1.2 Problem Description

The number of ECUs in a modern road vehicle has grown to a point where the complexity of distributed systems of ECUs in cars cannot be managed by an individual. This has an impact on the time taken to design vehicle electronics systems and a knock-on impact on the time taken to design, build and deliver the completed vehicle. This research addresses the issues surrounding the design of distributed systems of electronic control units and offers ways to

improve on reducing the number of ECUs, the mass of the vehicle, length of wiring, overall costs and time to market.

The many control algorithms and communications protocols that allow networks of connected ECUs to operate as a system of systems (SoS) are shared across many microcontroller (μC) boards to perform one of three tasks, [1] tasks that

a) generate signals, for example by reading sensors or controller area network (CAN) messages
b) compute output signals from input signals
c) use signals to command actuators or to create CAN messages

Sharing the control of separate features on a single processor or ECU reduces the number of ECUs required and therefore reduces cost.

Agile processes that determine the exact sharing of which features with which ECUs can be used to reduce the time taken to investigate candidate solutions, leading to a reduction in time to market for the completed vehicle.

Software simulations of the candidate hardware/software architectures allow for rapid prototyping that can build virtual systems of distributed ECUs with alternative network topologies and communications protocols, for example a star network with a central hub communicating with other star networks. These can be tested for speed of operation in desktop simulations and the capacity for ECUs to share other features can be calculated long before any system is physically built or any ECU algorithm programmed on to the target μC board.

Since the entire solution space is so large, any optimisation that relies on linear programming techniques or an exhaustive search of the space is either ‘non-deterministic polynomial (NP) hard’ or ‘NP-complete’ and unsolvable in polynomial time.

A heuristic method such as genetic programming (GP), a genetic algorithm (GA), simulated annealing and bin-packing algorithms are used for optimisation, such that a subset of the problem space is searched and a ‘best solution’ is obtained which may or may not be optimal but which will be the best of all the alternatives searched within a practical amount of allowed time.

Two case studies were conducted for this research, one a study of an ‘anti-lock braking system’ (ABS) and the other a study of ‘automated manual transmission’ (AMT) that examine the use of ECUs and the candidate architectures for processing, control and communication.

The research aim is to develop a process that will allow the rapid prototyping of distributed real-time embedded system models that can be executed for verification purposes, aided by these research objectives

1. Identify practices, methods and architectures in design and implementation
2. Implement models to represent vehicle electronics, ECUs, communications buses and constraints
3. Obtain and develop suitable application code and software patterns that implement the features and low-level device drivers
4. Automate the process to generate suitable executable models
5. Develop a tool for the allocation of features to the distributed system
6. Develop a suitable approach for the testing of the executable model

Novel aspects include a genetic algorithm that reduces the time taken to identify and examine suitable candidate solutions and the elimination of infeasible areas of the total possible search space by considering the geometric properties of the relations between non-zero elements of binary valued arrays and their representations in alternative number bases. Multi-dimensional plane figures and solids identified by this research contain the feasible mappings to ECU and feature pairings in their Cartesian co-ordinate points.

2. Critical literature review

The topics in the literature that are of primary relevance to the current research are embedded systems, communication methods in distributed embedded systems, the automated generation of code, the execution of models and genetic algorithms (GA). The following is a review of the literature and its influence on the work which followed in subsequent years.

2.1 Embedded systems

"An embedded system is an application that contains at least one programmable computer (typically in the form of a μC, a microprocessor or digital signal processor chip) and which is used by individuals who are, in the main, unaware that the system is computer-based." [2].

Embedded systems are becoming increasingly complex and sophisticated. This requires new approaches to design of SW development. The issue is outlined in Kashif *et al*. [3] where they propose the use of a specific UML tool to automatically generate code from a model based development and testing flow, for embedded systems, but they recognise that UML needs to be standardised in order for other model compilers to compile 'executable translatable UML' (xtUML) models. This research uses these models to communicate the systems requirements and to generate code for the vehicle features and controllers.

The reasons for new ECU architectures and interfaces are discussed in the literature by Rajnak *et al*. [4] and by Sander *et al*. [5]. It is because of the exponential growth in complexity of automotive electronic systems in the past decade which looks likely to continue. Possible solutions include 'virtualization' through the use of the Automotive Open System Architecture (AUTOSAR), an initiative aiming to standardize an open architecture for ECUs. The AUTOSAR standard was originally written for single core ECUs but guidelines and recommendations now provide support for multi-core architectures as per Becker *et al*. in [6]. This research aims to find architectures that reduce the number of ECUs.

Comparison of different processor designs is of relevance to this research. Embedded systems are traditionally deployed on single core processors which cannot parallelise tasks and which therefore employ time-slicing to manage multiple tasks in particular ways so as to achieve the appearance of running many tasks simultaneously. Dual and multi-core processors will also be considered for parallelism of tasks.

With the rate of increase in complexity, highlighted by Won *et al*. [7] of not only automotive electronics systems but electronic systems generally, developments in the design and architecture of such systems is necessary in order to manage the complexity so that individuals can understand, describe, design and maintain these systems and manage changes, with traceability and clarity. Complexity is managed in this research by handing over tasks to automated processes that maintain traceability and reusability.

2.2 Distributed embedded systems

One of the biggest challenges for the designers of modern vehicles is the management of the continuous increase in the number of ECUs in each vehicle, as discussed by Senthilkumar *et al*. [8]. In addition, advanced functionalities such as ABS, adaptive cruise control and climate control put higher computational demand on ECUs, which further increases the design complexity of automotive control systems. The actual number of ECUs in any single vehicle is quoted as anywhere from 70 to more than 100 in the literature [6], [9], [10], [11], [12].

The embedded systems in modern vehicles fulfil numerous tasks and functions and, due to the fact that several functions are distributed to more than one ECU, communication is a major issue during design and development. Communications are reduced as an outcome of this research by the reduction of the number of ECUs in the vehicle.

2.3 Communication networks for distributed embedded systems

Vehicle control networks, such as CAN, LIN and FlexRay [13], provide ways for ECU nodes to communicate but the growing number of ECUs in vehicles presents problems in relation to weight, size and complexity. Quigley *et al*. [14] describe a "cost modelling" process which results in a reduction of the number of ECUs and a smaller number of FlexRay networks to replace a larger number of CANs. Other systems which manage the same types of communications include 'Byteflight', which sits between CAN and FlexRay in terms of chronological history, LIN which is a lower cost and lower bandwidth than CAN and 'Media Oriented Systems Transport' (MOST) bus. No advantage has been found in using these over CAN in this research. A major concern at the outset of this research was to reduce the total mass of a vehicle by the reduction of the electrical wiring. Whilst this could amount for up to 50kg in some cases, the area in which most wiring could be saved is electronic communication wiring which is of less cross sectional thickness and mass than the electrical wiring used for power.

2.3.1 CAN and FlexRay

One of the major issues with ECUs in vehicles is that they are all interconnected via one or more communications buses and that they contest arbitration in order to be allowed to activate their respective features.

Created by Robert Bosch GmBH and first presented in 1991 [9], the controller area network (CAN) protocol is an electronic system of communication between µCs which uses a specific structure of packaged messages (referred to as frames) containing unique message identifiers and priorities which can be read by every other node on a bus constructed of a twisted pair of wires terminated at both ends by a 120 ohm resistor. With a communication speed of up to 1Mbit/s, CAN is slower than the alternative system, 'FlexRay', which can communicate at speeds of up to 10Mbits/s but CAN is simpler and more robust.

The CAN protocol is like a conversation around a dinner table or at a lively party. With so many people all having something to say, those wishing to add their own contribution must

wait for a gap or a lull, but if two people spot this lull and try to speak at the same time some sort of prioritising needs to be worked out to allow just one of them to continue talking. This decision as to who is allowed to speak is the basis of 'arbitration'.

Arbitration on CAN is contested one bit at a time until one node wins and continues sending its message. At the moment (clock tick) that any other message loses arbitration, it stops sending and goes into listening mode. This is described by Voss in [15] and by Wey *et al*. in [16] along with an alternative arbitration mechanism for CAN which allows faster message sending without any node taking control of the bus and preventing access to other nodes.

The choice of hardware for an embedded system of ECUs is affected by factors such as the ability to perform in large temperature ranges and possession of appropriate execution speed and memory resources. In the literature, there are examples of relatively inexpensive pieces of equipment being used to build experimental CAN control systems. Voss [17] describes the use of Arduino CAN bus shields to create simple two-node networks and, whilst Arduino is relatively inexpensive, it is not of industrial strength and would not be the first choice for finally implementing a system in a vehicle. It is however useful for modelling communications and control systems for the purposes of research and to demonstrate and validate models of hardware/software architecture and for some 'hardware-in-the-loop' (HILs) simulations.

Another issue with CAN is that it is used as a means to be able to continually add ever more features by simply introducing new nodes to the bus. In [18] Forsberg and Hedberg suggest that CAN nodes can be added to the network without the need to make any changes in hardware or software of any node and application layer. This may be true for instances where a new feature listens to other nodes' messages but does not need to be heard by any other node. This would be quite rare for a feature which communicates via the CAN bus, as the entire purpose of communicating between nodes is so that decisions can be made about whether to allow features to be activated based on their priorities over features currently accessing the bus.

For any existing feature which would necessarily be affected by the introduction of a new one, the programming code on the ECU for the current feature would have to be amended so as to include some sort of procedure or state-change which would be activated whenever the new feature gains access to the CAN bus.

A consortium of automobile manufacturers developed the FlexRay network [19] to provide and manage deterministic scheduling with a master network node and a speed of up to 10Mbit/s compared with CAN's asynchronous event triggered communication protocol with a maximum of 1Mbit/s. Because of the higher bandwidth and greater flexibility of FlexRay over CAN, and because of CAN's simplicity and large number of users who understand the system, a combination of CAN and FlexRay would be beneficial to the speed and simplicity of vehicle ECU communications systems, over one consisting entirely of FlexRay. This research has used CAN exclusively for its output but recognises the importance of alternative communication protocols and for future research into these.

2.3.2 AUTOSAR

AUTOSAR is an open and standardized automotive software architecture, jointly developed by vehicle manufacturers, suppliers, and tool developers. Its main goal is to introduce a standardized layer between application software and the hardware of an ECU. Thus, the software is largely independent from any chosen µC and vehicle OEM, making it reusable for several individual ECU systems.

In order to integrate AUTOSAR software Components into a network of ECUs, AUTOSAR provides description formats for the complete system as well as for the resources and configuration of the single ECUs. These descriptions are kept independent of the software Component Descriptions.

AUTOSAR defines the method and tool support needed to bring the information of the various description elements together in order to build a concrete system of ECUs. This includes especially the configuration and generation of the Runtime Environment and the Basic software on each ECU.

In order to fulfil the goal of transferability, AUTOSAR defines a layered software architecture and a formal description language for software Components so that these components can be implemented independently from the underlying hardware. Due to this abstraction a virtual function bus (VFB) can be used to assemble and integrate these components to a virtual AUTOSAR system and to verify the consistency of the communication relationship between software components. In a vehicle network this approach allows vehicle manufactures to break down the complexity of their systems in a very early design phase of the product development cycle.

Compared to traditional development processes AUTOSAR does not only simplify the exchange of software Components, it also provides a method to handle and manage the increasing complexity of vehicle systems. In online literature [20] the VFB is described as the sum of all communication mechanisms and essential interfaces to the basic software provided by AUTOSAR on an abstract, that is technology independent, level. When the connections between AUTOSAR software Components for a concrete system are defined, the VFB will allow a virtual integration of them in an early development phase. AUTOSAR reflects broad acceptance of a component paradigm within the automotive industry. Schreiner and Goeschka [11] have sliced the FlexRay stack in AUTOSAR's communication stack to achieve software with a 30% smaller memory footprint and a runtime performance of 10% less central processor unit (CPU) usage.

2.3.3 Wireless communication

The communications systems researched so far all rely on signals being sent and received along (predominantly) copper wire, which is heavy in the amounts that make up the entire communications bus for a vehicle, maybe as much as two miles of wiring weighing 50lbs according to Laifenfeld and Philosof [21] who state that this could be reduced by the use of

remote input/output systems (’remote i/o’) such as the Vehicular Wireless CAN (ViCAN). ViCAN combines wireless and traditional wired CAN physical layers with the aim of reducing wiring complexity. ViCAN also uses off-the-shelf components such as the MCP2515 CAN controller, present on the Arduino CAN shields. This, at least, should mean that Arduino would make for a suitable simulation of ViCAN if wireless sensor/transmitter shields can be sourced for the Arduino board. There is a subsequent issue with the cross sectional area of cabling used in 12V vehicle electric systems which could be reduced along the entire length of cables (further reducing the total weight of the vehicle) by the introduction of higher voltage systems (most likely 48V).

2.4 Automatic code generation

Since one of the ways to speed up the development of software is to automate the code generation from requirements documents, it is worth considering the tools which can or cannot achieve this and those which can generate code from diagrams. This research examines methods of presenting systems requirements in a way that is readable by both humans and machines so that communication of ideas between engineering teams and individuals is achievable and so that the tools used in this communication can also be the tools that design the system.

The nature of UML as a semi-formal modelling language, its ability to validate correctness of embedded software development and its lack of accurate semantics, is questioned in the literature by Guo *et al*. [22]. Ambiguities in the models lead to vague interpretations which make automation hard. Model transformation is proposed, between UML and Simulink in embedded software design, using ‘Atlas Transformation Language’ (ATL), compared with the Object Management Group’s (OMG’s) Meta Object Facility (MOF) which Guo *et al*. claim is not as simple or as practical as ATL. The area of ‘meta-modelling’ is worthy of investigation as a means to achieve some automation of code for embedded systems.

As an alternative to deployment on µCs, however, automated code generation from UML to VHDL for FPGA has been proposed by Moreira *et al*. [23] and this could be useful, if the transformation rules can be developed further. The specific implementation of a “…smart component.” is not presented in detail and shortcomings are acknowledged. The conclusions claim that the results show rules applied to UML classes, attributes and behaviours, are sufficient to develop “…simple systems.”, although VHDL structures are not currently generated.

Another potential method of automation is to take specifications or a requirements document in natural language, where it might be possible to extract class diagrams by applying natural language processing (NLP) according to Ibrahim and Ahmad [24]. This could partially automate the software design process although, at present, some class relationships cannot be identified as being ‘one to one’, ‘one to many’ or ‘many to many’. Since UML tools can produce skeleton object-oriented code (‘stubs’ in ‘C’ source code) from class diagrams, it would be useful to be able to create the diagrams themselves from natural-language requirements-documentation, thereby closing the gap between specifications and full

implementation. If this is achieved at both ends of the development lifecycle in parallel, the possibility of automating the complete lifecycle can be ushered in more quickly.

Hakan Burden's thesis on xtUML [25] claims that it is one of the easiest UML tools to learn how to use, backing this up with results of experiments on students learning to use the tool for the very first time. The executable and translatable nature of this variant of UML means that class diagrams and state charts can be used to generate functioning source code in 'C/C++' and 'JAVA' which will compile into executable files on the Windows operating system (O/S). Using BridgePoint's proprietary action language, code can be attached to the state machines which form the executable part of the language and xtUML's integrated development environment (IDE) will translate this into one of three software language paradigms before then compiling that source code into an executable ('.exe') file to run in a Windows console. On further investigation of xtUML, the research has found that there are shortcomings with respect to generating code for embedded systems, which are addressed elsewhere in this thesis.

From their paper on quantifiable software deployment [26] Hughes and Lovstad state that "…the deployment aspect of UML is relatively underdeveloped." and that use of the 'Structure and Performance' (SP) paradigm can either enhance the UML deployment diagram, with 'Object Constraint Language' (OCL) constraints - or simply replace it. However, this is once again an example of a case study based on a relatively simple database system which does not involve hard real time scheduling and deadlines, typical of embedded systems. Whilst it may enhance the deployment aspects of UML, it does not improve its capability to model real time systems.

'Model Transformation' is the process of converting one model to another model of the same system as described in the literature by Miller and Mukerji [27]. This is of interest in terms of either translating requirements documents into specifications or highly abstracted diagrams into source code templates or executable programs. Transformations between models should be possible by use of the mathematics of graph theory and set theory, where a function or an algebra maps one graph to another. The purpose is to be able to see a UML model as a graph and to translate this into another graph representing, for example, a Simulink model/simulation. An explanation of how this is done is attempted but not fully explained in papers by Brisolara and Wagner [28]. Their methods of specifying the grammar of these graphs leaves a lot unsaid, particularly the nature of the "…algebra…" which is applied to the initial graph to arrive at what they call an "…attributed graph". They begin with very complex models rather than simple ones and they show few examples of an actual transformation or of an algebra which performs this, or how the results are achieved with an 'end-to-end' example.

Many UML tools request the user to select a programming language upon creating a new project. Ideally, models would be created and saved with no specified programming language in mind and could be translated later into any language without changes to any part of the UML diagrams or action language code. Models are programming language independent,

according to Romaniuk in [29] who identifies the problem that software engineering which focuses on models can often miss the point that computers need programs. Thus, a translator is proposed which can translate UML state machine descriptions to modules in a programming language. It is recognised that there are very few UML code generation tools which are aimed at embedded systems or which can transform behaviour descriptions.

Cherif, in [30] proposes the UML 'Modeling and Analysis of Real Time Embedded Systems' (MARTE) profile as the method for modelling reconfigurable 'systems-on-chips' (SoC), highlighting many of the issues already raised elsewhere in this thesis (reducing design complexity, development costs, time to market etc.) despite being aimed predominantly at implementation on FPGA. It is even suggested that model transformations can be carried out to generate executable models from higher level models, although this is not demonstrated and the conclusion is that MARTE lacks concepts for the complete specification of reconfigurable SoCs.

The UML MARTE profile is one of many UML extension mechanisms defined in different industrial and research contexts. However, most of these extensions which cover a wide range of applications suffer a lack of standardization. This misalignment on to the idea of a standard language is very problematic since it discourages and limits their industrial exploitation according to Demathieu in [31].

Chen *et al*. [32] recognise an essential deficiency of UML is that it standardizes the syntax and semantics of diagrams, but not necessarily the detailed semantics of implementations of the functionality and structure of the diagrams in software. They call for and demonstrate the construction of a new UML profile for modelling embedded system platforms, in which layers and wrappers provide platforms for high level applications.

In [33], Bazydlo *et al*. propose a method for using UML state machine diagrams exported from Rhapsody or Sparx into Extensible Markup Language (XML) to generate an effective program in 'Hardware Description Language' (HDL) such as Verilog. Exporting of UML into 'XML Metadata Interchange' (XMI) files for the purpose of interrogation by ad hoc or bespoke application code is investigated in this research.

Pop *et al*. [12] deal with the architectures, communications protocols, heterogeneous distributed real-time embedded systems and design optimization all of which form major areas of research in this thesis.

This is possibly one of the most important papers so far discovered in the existing literature around the research topic as it becomes more narrowly defined for the second and third years.

2.5 Executable embedded system models

A HILs for ABS, shown by Park *et al*. in [34], is intended to solve the problems of reducing the development time and costs associated with ECU design. An MCP5554 is used to support 2 channels of FlexRay, 4 of CAN, 2 of LIN, 24 'analogue to digital converter' (ADC) channels and 8 'digital to analogue converter' (DAC) channels. Control logic is designed

with the use of graphics-based programming languages such as Simulink or LabVIEW and the conclusion is that an ABS HILs was performed to confirm that a rapid control prototype (RCP) is suitable for use in developing vehicle ECUs. Since a case study of an ABS simulation is intended for the methods and experimental aspect of this research, this is highly relevant. software Architecture for ECU of AMT [35] describing five layers and two bases would be equally applicable to the ABS case study.

Hammer *et al*. acknowledge that "While much progress has been made in general-purpose object-oriented methods, the construction of object-oriented distributed real-time systems still contains many challenges." [36]. This highlights an ongoing problem with object-orientation as a means to producing executable models for real time embedded systems.

A tutorial example of xtUML from BridgePoint successfully generates code which imitates/simulates the order/time-sequence of the events and actions necessary to cook food in a microwave oven, from opening the door and inserting the food to closing the door and setting the timer. The output from model compilation of xtUML is a set of '*.c' and '*.h' files (or another alternative choice available to the developer from C++ or JAVA) which can be further compiled into machine code using either gcc, cgWin, VisualStudio or other C/C++ compilers (Bloodshed *et al*.). Once compiled, the generated executable program produces lines of print output to console describing the states and actions of the microwave cooking process with timed sequences.

This is not code which can immediately be implemented or deployed on a μC. It is code, generated from BridgePoint's action language, inserted into textual representations of state machines for the sole purpose of reporting on the sequence of desirable actions which are not captured in the usual UML diagrams (use-case diagrams, class diagrams, state machines etc.) and so far it runs on Windows and has not been flashed to a μC.

A printed version of the output, which was redirected to a text file, has been reproduced in the Appendices section of this thesis, for the purposes of illustrating the limitations of xtUML in generating useable code outside of the desktop environment in which it was created.

Although there has been related work on the uses object-based design of embedded software using real-time operating systems since the early 1990s [37] there is still a long way to go before UML tools and object oriented methods can produce real-time software systems for distributed embedded ECU/μC systems such as those which form this research or which would be of benefit to vehicle manufacturers for deployment in their vehicles. This research aims to address that problem with the use of object-oriented design tools such as UML to model software systems and hardware architectures.

2.6 Resource allocation

In this research, resource allocation problems refer to finding solutions that place known quantities of executable code on devices with sufficient program storage and runtime memory to allow them to perform as expected on an optimal number of ECUs. Many combinatorial

optimization problems such as the bin packing and multiple knapsack problems involve assigning a set of discrete objects to multiple containers. Fukunaga and Korf [38] propose several improvements to bin-completion that significantly improves search efficiency. They also show the generality of bin-completion for packing and covering problems involving multiple containers, and present bin-completion algorithms for the multiple knapsack, bin covering, and min-cost covering (liquid loading) problems that significantly outperform the previous state of the art. However, they show that for the bin packing problem, bin-completion is not competitive with the state of the art solver.

In the literature other types of resource allocation problems include scheduling, that is the queueing and execution of algorithms and processes sequentially and/or in parallel so as to optimise (minimise) the total time taken to complete all of the processes and tasks.

The problems of resource allocation are being considered on a widening scale in various branches of operations research, management science and optimization theory. This follows from the need created by practical applications concerning the management of scarce resources like manpower, machines, materials and funds in some technical, economic or social complex systems. There also occur some specific applications, for example, problems of resource allocation in computer systems. In consequence, there have arisen a variety of models for the above problems, approaches to their solution and a diversity of terminology and interpretation of results. Such a situation prevents perceiving the state of the art across the breadth of the problem and causes difficulty in finding desirable directions for further research.

In [39] Bar Yehuda *et al.* present a general framework for solving resource allocation and scheduling problems. Given a resource of fixed size, algorithms that approximate the maximum throughput or the minimum loss by a constant factor are presented. Many problems are addressed by the approximation factors applied to them, among which are: (i) real-time scheduling of jobs on parallel machines, (ii) bandwidth allocation for sessions between two endpoints, (iii) general caching, (iv) dynamic storage allocation, and (v) bandwidth allocation on optical line and ring topologies.

Fleischer and Wahl study the classical problem of scheduling jobs online on m identical machines without preemption, i.e., the jobs arrive one at a time with known processing times and must immediately be scheduled on one of the machines, without knowledge of what jobs will come afterwards, or how many jobs are still to come. In [40] they state that whilst scheduling problems are of great practical interest, even some of the simplest variants of the problem are not fully understood and they go on to present a new online algorithm, MR, for non-preemptive scheduling of jobs with known processing times on a number of identical machines (m). It beats the best previous algorithm for $m \geqslant 64$. For $m \rightarrow \infty$ its competitive ratio approaches <1.9201. In their paper they propose another small improvement on the upper bound of the competitiveness of the online scheduling problem, decreasing it to <1:9201 . For m>64 we beat the best previous bound of 1.923

The goal is to achieve a small makespan which is the total processing time of all jobs scheduled on the most loaded machine. Since the jobs must be scheduled online we cannot expect to achieve the minimum makespan whose computation would require a priori knowledge of all jobs (even then computing the minimum makespan is difficult, i.e., NP-hard). The quality of an online algorithm is therefore measured by how close it comes to that optimum. It is said to be *c*-competitive if its makespan is at most *c* times the optimal makespan for all possible job sequences.

In a seminal work of 1966, Graham [41] showed that the List algorithm which always puts the next job on the least loaded machine is exactly (2 − 1m)-competitive. Only much later were better algorithms designed. In the same body of work, Graham shows the anomaly of sharing resources to parallelise processing that can actually have the effect of increasing the amount of time taken to process a fixed set of tasks.

Resource allocations problems are generally NP Hard. The Quality of Service (QoS) based Resource Allocation Model (Q-RAM) problem of finding the optimal resource allocation to satisfy multiple QoS dimensions is NP hard according to Rajkumar *et al*. [42] who presented an analytical approach for satisfying multiple quality of service dimensions in a resource constrained environment. Using this model, available system resources can be apportioned across multiple applications such that the net utility that accrues to the end users of those applications is maximized. Several practical solutions to allocation problems are beyond the limited scope of Q-RAM.

In [43] Katoh and Ibaraki describe the resource allocation problem seeks to find an optimal allocation of a fixed amount of resources to activities so as to minimize the cost incurred by the allocation. A simplest form of the problem is to minimize a separable convex function under a single constraint concerning the total amount of resources to be allocated. The amount of resources to be allocated to each activity is treated as a continuous or integer variable, depending on the cases. This can be viewed as a special case of the nonlinear programming problem or the nonlinear integer programming problem.

In [44] Ramanathan and Ganesh, Multi-criteria resource allocation (mcra) problems involve allocation of limited resources to different activities keeping in mind many conflicting criteria. They have been effectively solved using multi-criteria decision making (mcdm) techniques. The Analytic Hierarchy Process (AHP) has emerged as a useful decision making technique for solving mcra problems.

Whilst these are all of interest to this research and to further problems of resource allocation and scheduling that may be investigated in the future, they cover a topic beyond the scope of this research with respect to the allocation of the kinds of resources that this research has been interested in, specifically assigning single features to an ECU that will execute the algorithms of a single control process or that will share the processing of multiple algorithms in a predetermined specified order that is neither optimised nor parallelised.

2.7 Heuristic Methods

"In recent years, a large catalogue of heuristic techniques has emerged inspired by the principle that satisfaction is better than optimization, or, in other words, rather than not being able to provide the optimal solution to a problem, it is better to give a solution which at least satisfies the user in some previously specified way, and these have proved to be extremely effective…

…The term *heuristics* comes from the Greek word "heuriskein'', the meaning of which is related to the concept of finding something and is linked to Archimedes' famous and supposed exclamation, Eureka!''. Verdegay *et al.* [45]

Heuristic methods are those which seek to solve large scale problems that are, in complexity theory, NP hard or NP complete with a preference for speed of execution and without concern for optimal solutions. This research relies on heuristic methods to search the problem space in the allocation of ECUs to features and to reach solutions in a shorter time.

Lo [46] offers heuristic algorithms for solutions to 'task assignment' problems in distributed systems.

Among well-known and well documented heuristic methods are;

Swarm Intelligence, Tabu Search, Simulated Annealing, Genetic Algorithms, Artificial Neural Networks, Support Vector Machines, Bin Packing and specific examples of solutions to the Travelling Salesman Problem (TSP) for example the multiple travelling salesman problem in [47] which can be applied to the networks of cabling with in a vehicle to use the least amount by following the shortest route –particularly in redundant systems where the wiring is duplicated via an alternative route.

2.7.1 Genetic Algorithms

"Genetic Algorithms (GAs) are adaptive heuristic search algorithms premised on the evolutionary ideas of natural selection." [48]. They are algorithms that evolve in ways that resemble natural selection and can solve complex problems even their creators do not fully understand…When sperm and ova fuse, matching chromosomes line up with one another and then cross over partway along their length, thus swapping genetic material. This mixing allows creatures to evolve much more rapidly than they would if each offspring simply contained a copy of the genes of a single parent, modified occasionally by mutation. [49]. This swapping of genetic material is simulated for non-biological systems to provide solutions that evolve using cross over and mutation. Initially the neighbourhood search operators (crossover and mutation) are applied to the preliminary set of solutions to acquire generation of new solutions. [48]

Although only one pair of parents and a single child are required to generate successively improved generations, traditional GAs use at least two offspring and can have many 'families' in the population, all of which compete to survive and reproduce.

The effects of population size are outlined in [50] [51] [52] [53] [54] but for the purposes of this research, no advantage was found in increasing the population size beyond two parents and two children in any generation.

Aleti considers 'Component deployment optimisation' [55] as part of a wider problem of "Genetic Algorithms for design of embedded systems"

The Component Deployment Problem (CDP) refers to the allocation of software components to the hardware nodes, and the assignment of inter-component communications to network links, very much like the ECU/features problem in this research and the connections of nodes via CAN. This is discussed in the ABS case study later in this thesis.

2.7.2 Simulated Annealing

Simulated annealing is an alternative to GAs for categorised solutions and, although this method has been researched for this thesis, no advantage was found in using it for this specific problem, despite the discrete nature of the solutions and the fitness function.

Kirkpatrick *et al.* [56] use simulated annealing as a tool for solving the travelling salesman problem and another of wiring electronic systems. Whilst these share some common features with the ECU problem, by their own admission, "Heuristics are rather problem-specific: there is no guarantee that a heuristic procedure for finding near-optimal solutions for one NP-complete problem will be effective for another".

Zeinelden's 'improved simulated annealing' (ISA) in [57] offers a solution to constrained optimization problems but there still remains an issue of lack of any discernible relation between neighbouring solutions in the discrete case, specifically in the ECU/features problem. For these reasons, simulated annealing was not adopted as the heuristic of choice for this research.

A test case using simulated annealing for a specific problem with a known global optimum was investigated to ascertain its usefulness and ease of implementation. Using the knapsack problem that is described later in this thesis, both a genetic algorithm and a simulated annealing algorithm were programmed to find a solution near to the known optimum for a given set of variables. In tests, the GA found the known solution 74% of the time after taking 1.3 seconds to execute 10000 individual runs. For the same problem, the simulated annealing approach managed a success rate of 11% but took only 0.385 seconds to execute 10000 runs.

The code for the simulated annealing algorithm, which follows El Ghazali's example in [58] is included here

Input: Cooling schedule.
s = *s*0 ; /* Generation of the initial solution */
T = *Tmax* ; /* Starting temperature */
Repeat
    Repeat /* At a fixed temperature */

Generate a random neighbor s' ;
$\Delta E = f(s') - f(s)$;
If $\_E \leqslant 0$ **Then** $s = s'$ /* Accept the neighbor solution */
Else Accept s' with a probability $e^{\Delta E/T}$;
Until Equilibrium condition
/* e.g. a given number of iterations executed at each temperature T*/
$T = g(T)$; /* Temperature update */
Until Stopping criteria satisfied /* e.g. $T < T_{min}$*/
Output: Best solution found

Whilst this is much simpler to code than the GA and takes less time to execute for the same problem, because it has only one strand of DNA in memory at any time, it was unwieldy and difficult to tune to ensure accuracy for the specific knapsack problem. This further informed the decision to proceed with a GA approach for the resource allocation problem specific to this research.

2.7.3 Bin packing algorithms

One-dimensional bin packing algorithms, described by Martello and Toth [59] are considered in this research as ways to solve problems of initial constraints on size and capacity of features' requirements and ECUs' properties. Considering each ECU as a container, with a specific and finite capacity for memory (for example), the memory requirements of the features can be seen as items of a given size that need to be placed in the containers. More than one feature may be assigned to a single ECU if there is sufficient remaining capacity after one or more features have been allocated. With no necessity to find an optimal solution with respect to cost of these placings, a bin packing algorithm can be used to provide a starting position for the GA, with candidate solutions already suggested by the bin packing algorithm. The ECU/features problem with respect to capacity of a single variable is analogous to a pipe cutting problem in which a plumber is required to cut specific lengths of pipe from a variety of pipe lengths available with minimum waste.

When there is only one container and items are selected so as to yield the greatest total value (with items assigned arbitrary quantitative values) this is known as the knapsack problem.

The types of bin packing algorithm available, described by Martello [59], include Next-fit (NF), Next-fit Decreasing (NFD), First-Fit (FF); First-Fit Decreasing (FFD); Best-Fit (BF) and Best-Fit Decreasing (BFD) and the worst case performance bounds for these simple one dimensional packing algorithms suggest that both FF and BFD is suitable for the purposes of this research.

The 'first fit' algorithm seeks to fit items in containers (by constraints on weight [mass], volume, etc.) by placing them, in turn, in some arbitrary order to the first available container with sufficient remaining capacity. This bin will continue to be used for further items until an item is found that it cannot accommodate at which point a new/empty bin will be allocated,

that is, if an item fits in bin one, it should be placed there. If not, bin two should be examined and so on.

The ‘first fit decreasing’ algorithm employs the same method as the ‘first fit’ algorithm but sorts the items from largest to smallest before picking them to be placed in any container.

Yue provides a simple proof of the inequality for the FFD bin packing algorithm [60] such that the algorithm never allocates more than 11/9 of the optimal solution + 1.

Traditionally, the first fit decreasing algorithm deals with containers of similar sizes but can be applied to problems with containers of differing sizes, where both the items and containers can be ordered before any packing occurs.

Alahmadi *et al*. propose an enhanced first fit decreasing algorithm [61] for scheduling and allocating tasks to virtual machines in cloud based data centres.

2.8 Summary

Currently, there exist methods and/or tools to automatically generate various stages of the design lifecycle. Where these are well developed, there is little novelty. Elsewhere, the research seeks to find solutions to simplify the design of distributed embedded real-time systems. The literature indicates that UML is a problematic tool for the automated generation of fully executable code in an embedded real-time system and therefore the research will seek to use this tool only for modelling at high levels of abstraction or for the purpose of describing use-cases and static class structures and attributes. UML diagrams are used to export XMI files, where possible, and to generate some code structures which can be manually completed with attributes and methods in programming language source code.

Because of the drive in the literature for new ECU architectures, these have also been researched further. As shown in earlier sections of this literature review, although the need for these new architectures is acknowledged and identified, no end-to-end examples have been found which clearly demonstrate methods or processes for designing these new architectures in ways which optimise the available resources for performance and ease of understanding and reusability. Therefore, the needs for reduction in complexity of embedded ECU systems and the management of the increase in numbers of ECUs, prevalent in the literature, still needs to be addressed as an exercise in designing and automating the design of embedded systems.

The literature directs this research towards executable models, software patterns and the creation of a tool for the automation of design-space exploration. Such a tool would bring together existing methods for designing distributed real time embedded systems for the automotive industry and allow for a much quicker investigation and comparison of design alternatives.

The amount of computing time necessary to perform a single run of the GA, for any given problem relating to this specific research is not reduced by increasing the population size

unless the problem solving algorithm is parallelised. For a sequential, serial, algorithm that uses only a parent pair and a child pair, longer execution times serve the same purpose as increasing the population size, since the only real benefit is to generate more candidate solutions that still need to be examined sequentially.

In addition to the GA designed to solve the ECU/features allocation problem, a one-dimensional bin packing algorithm is employed in this research to provide a starting position for the GA, with candidate solutions already suggested by the bin packing algorithm.

The systems solution takes the topics from the literature and brings them together in a novel way to produce an end to end design of a vehicle electronics system, verified and validated in HILs. From requirements documentation to a hardware/software system, implemented on Arduino experimental boards with CAN bus μCs, this research documents the system requirements in UML and automatically generates C code that can be manually amended to produce the source code for controller software to be integrated with header files and drivers for the CAN bus communications protocol.

3. Proposed solutions to the ECU problem

The ECU problem is defined for this research as one of allocating features of a vehicle to one or more ECUs by the use of computer algorithms for the purpose of reducing the number of ECUs used and reducing the total cost to purchase. Existing methods have been built upon by this research to create novel ways of delivering candidate solutions to this problem.

This research delivers a 'brute force' algorithm that examines every possible permutation of allocating any number of features to the same number of ECUs or fewer. The algorithm includes numerical representations of the solution that cannot be realised because of conflicts in the sharing of resources. These need to be filtered so that only feasible solutions are considered. To this end, a refined version of the algorithm determines a subset of the problem space that eliminates infeasible numerical representations and greatly reduces the time taken to execute the algorithm whilst still finding the optimal solution.

A bin packing heuristic developed from [59] by this research has been programmed in software to quickly find a solution to the ECU problem. It is deterministic and, whilst it may or may not find the optimal solution, it will always deliver the same solution to any problem presented in the same way. Another problem-solving algorithm that has the potential to find the optimal solution but which may take longer to run is the genetic algorithm (GA) that models chromosomes, similar to those in biological reproductive systems and produces offspring that are genetic improvements on the previous solutions found by the same GA. The GA developed by this research is employed to search beyond solutions offered by the bin packing algorithm. Quantitative properties of the ECUs and features are examined by the GA to allocate ECU resources to the features' requirements in more efficient ways.

Computer based numerical methods in software are limited by the speed and memory of the available hardware. This research has extended the possibilities for counting beyond the maximum integer values of 32 bit arithmetic by use of alternative number bases. This then allowed larger problems to be solved by the brute force method, in order to verify the GA, whilst the execution times for the GA itself are reduced by the introduction of a variable limit on the number of iterations allowed before finding a better solution or terminating.

This research solves a problem that is not defined by a continuous function. GAs are particularly good where solution scores are not related to those of neighbours. The constant ranking and pruning of the population eliminates poorly scoring chromosomes. Generally, keeping the best chromosomes from the previous generation unaltered (as well as some mutations of it and children of it through crossovers), the new generation scores at least as well as the last.

Each ECU in the vehicle has a finite capacity to support the features. Software obtained from third parties and designed as output of this research has a finite need for ECU resources and these are tested by the GA before allocating and sharing ECUs and features.

3.1 Definition of the ECU problem

The ECU problem, addressed by this research, is an optimisation problem concerning the allocation of ECUs to features of a vehicle based on one or more variables such as, but not exclusively, software program storage capacity (memory or ROM) on the ECU(s) and program storage requirements (code size) controlling and operating the feature(s).

For the specific case of an algorithm to allocate memory to features on the vehicle, an example of a system of 20 features is provided with a distribution of requirements, arbitrarily chosen to highlight the possibility of either allocating one ECU to each feature or sharing multiple features on single ECUs. One example which allows for this is shown in figure 3-1, which takes just one variable into account – memory, as required by the feature(s). The arbitrary units of the y axis (kB) show examples of the memory that could be required in order to operate correctly.

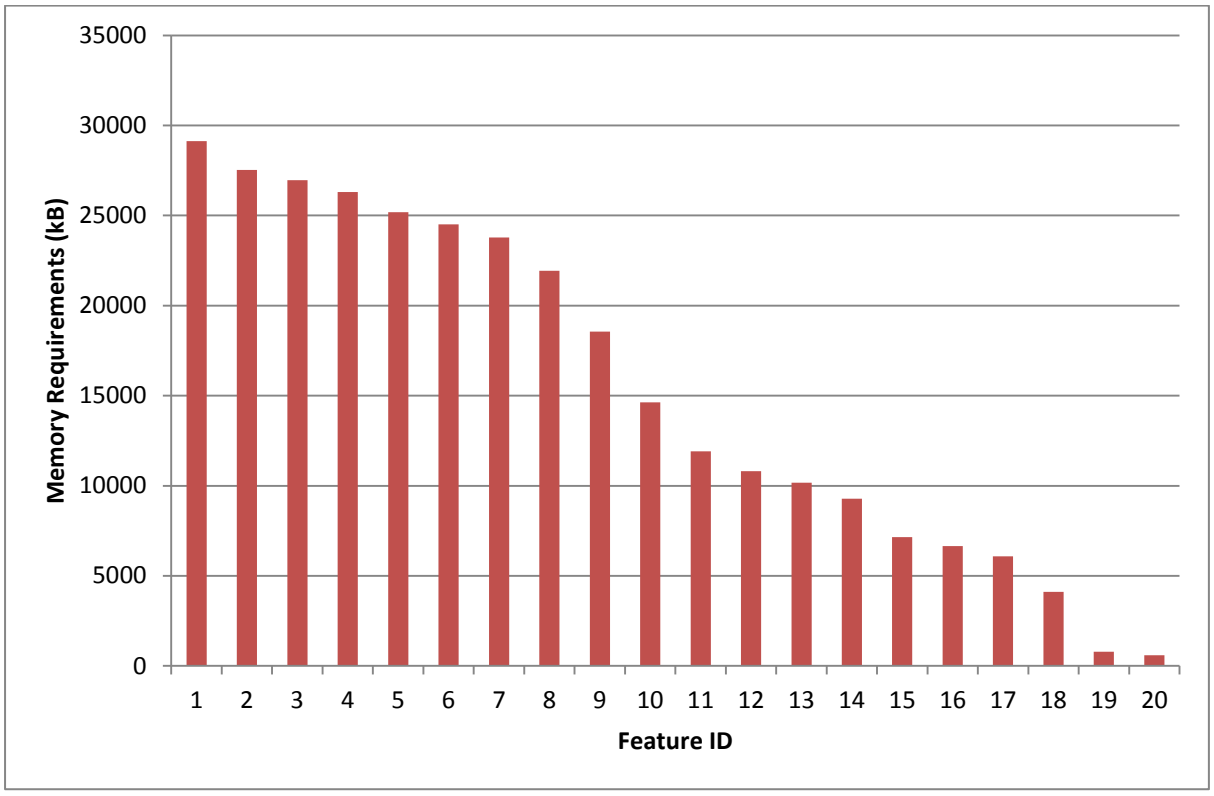


Figure 3-1 : An example of the memory required by 20 unique features of the vehicle

The total memory capacity of all the ECUs, tasked with supporting the features, needs to be greater than the combined total of the features' requirements and there should be provision for a 'worst case' solution in which every feature can be supported and every ECU can support at least one of the features giving a 1 to 1 ECU/feature relationship. Figure 3-2 provides an example of 20 different ECUs that might be available to support the features from figure 3-1. Again, they are arbitrarily chosen to illustrate an example where a 1 to 1 mapping is possible as well as the allocation of multiple features to single ECUs.

For example, the capacities of the available ECUs should accommodate the features with at least the possibility of one feature per ECU. The “ECU/Features Allocation” graph in figure 3-3 shows one such scenario. The values are those from the features and ECUs in figures Figure 3-1 and Figure 3-2 juxtaposed to show that every feature can be supported on just one ECU each.

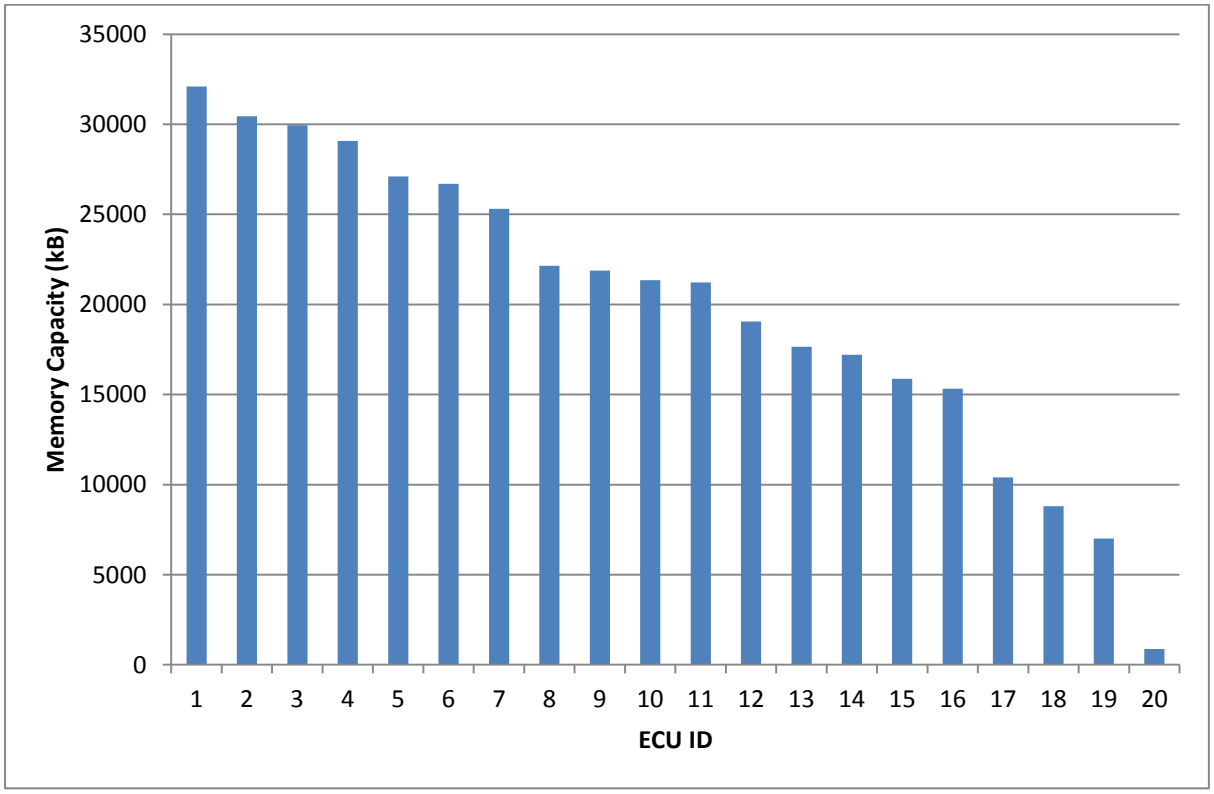


Figure 3-2 : Example memory capacities available for the 20 features on 20 different ECUs

In this research, the problem is solved by use of mappings represented by *n* x *n* arrays (square matrices) with 0-1 binary elements that assign one or more features to a subset of the available ECUs.

In the worst case, the solution arises from examining every possible pairing of ECUs and features and scoring each one by some function of the variables to be optimised. Because this ‘brute force’ method examines scenarios where a single ECU may be allocated many features as well as the infeasible scenario of single features being assigned multiple ECUs, it will be referred to in this research as the “Exhaustive Overkill” method.

A refined version of this will determine a subset of the total number of possible arrays that can be created, eliminating arrays whose values could place any single feature on more than one ECU. Whilst it is possible to share functionality of a single feature across more than one ECU, a problem that requires or allows this would consider that specific functionality to belong to two distinct features from the start. This solution method will be referred to as the ‘Exhaustive Search Program’ or ‘ESP’.

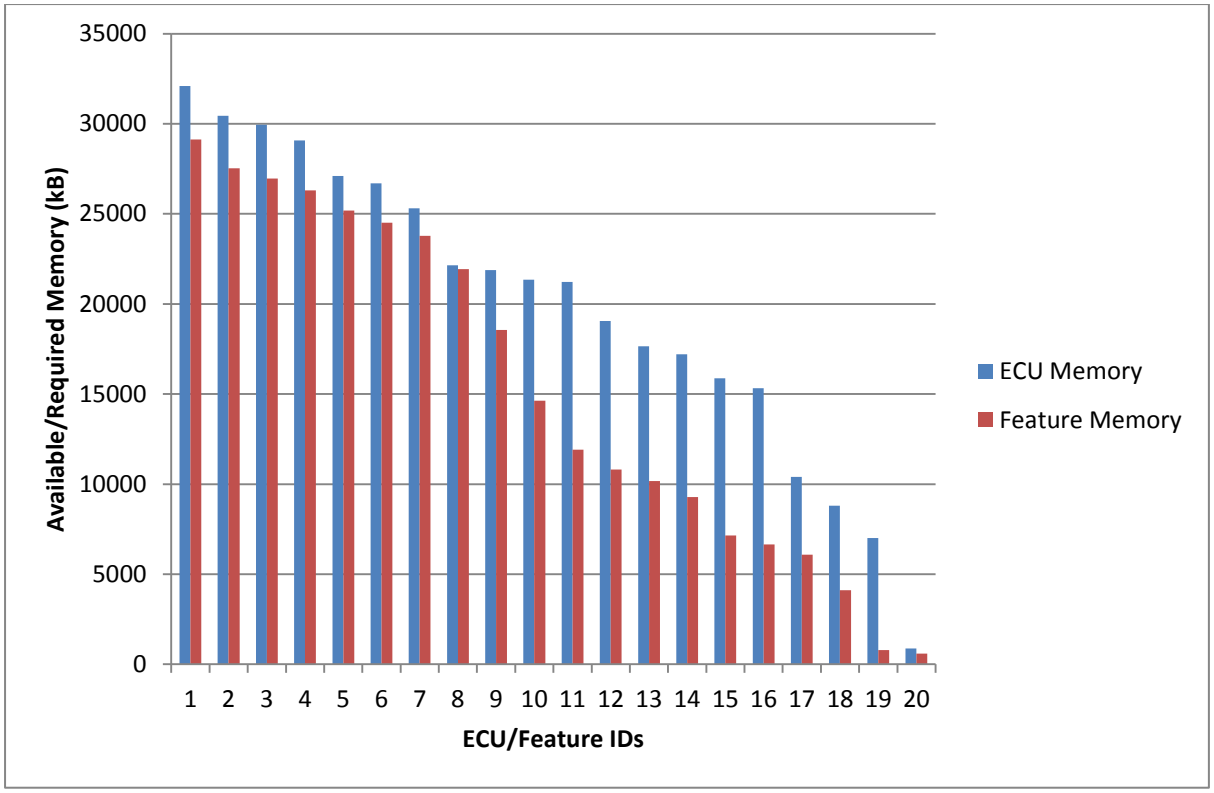


Figure 3-3: Example 1:1 allocation of ECUs to features ensuring that each feature is supported

Two heuristic tools were created by this research to solve the problem of allocating ECUs to features of the vehicle. The purpose was to find better solutions than those that might be, resources, skills and abilities, by searching the solution space as defined by a mathematical representation of the problem in the form of a two-dimensional array or matrix.

Established methods of optimisation by genetic algorithm (GA) and by an approach to the 'bin-packing' problem were employed alongside novel interpretations and implementations of GAs. The first consideration of the optimisation tool is whether the required features can be supported by a 1:1 mapping using a single ECU to support exactly one feature as in figure 3-3. If they can, a secondary consideration is whether they can all be supported using fewer ECUs.

It may be that the features cannot be supported on individual ECUs in which case the optimisation tool should determine whether they can be supported by sharing ECUs across multiple features, although it would be advantageous if the primary input to the solution tool were such that each individual feature had at least one ECU that could support it individually. One benefit of this is that features and ECUs can be developed alongside each other so that before integrating them into a vehicle, a solution already exists whereby the engineers working on separate 'stove-piped' projects have an ECU that supports their feature of interest even if no further optimisation is attempted to reduce the number of ECUs in the vehicle.

That is, it may be that a configuration should initially be sought whereby it is known before the optimisation program is executed that each feature has an ECU that can support that feature if required to do so individually. This could be done manually by an engineer or it could be a phase of the automated optimisation that is a check before trying to reduce the number of ECUs needed. It would be an easy matter for 100 engineers to select 100 ECUs to support 100 features – even if they were instances of the same ECU by design, manufacturer, capacity, cost, etc..

For the simplest, trivial, problem that could be solved by the optimisation tool, two features, supported by a maximum of two ECUs would be represented by the 2x2 identity matrix in which the starting state would be the identity matrix

$$C = I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

to represent feature number 1 supported on ECU number 1 and feature 2 supported on ECU number 2 (in the implementation of this in the 'C' programming language in this research, these are actually features 0 and 1 supported on ECU 0 and 1 respectively). In a genetic algorithm, this matrix is represented as the vector (C_{11}, C_{12}, C_{21}, C_{22}) and is referred to as a chromosome (hence the variable name, "C") because pairs of these are spliced and mutated to produce new chromosomes, similar to the process of biological reproduction, whereupon the suitability of each chromosome is tested and scored to determine whether it will survive and reproduce further generations. The testing and scoring in the biological world is performed by natural selection and survival of the fittest.

In any representation of this ECU/features problem, the total number of feasible chromosomes (those with exactly one non-zero entry, the number 1, in every column) is n^n. The proof is that for each column of an *n* x *n* matrix there are *n* unique positions for the non-zero entry. This multiplies by *n* for every column so that the function determining the number of unique feasible chromosomes is

$$f(n) = n_1 \text{x } n_2 \text{ x } \dots \text{ x } n_n = n^n \qquad (3\text{-}1)$$

and for the general case of an oblong *n* x *m* matrix

$$f(n) = n_1 \text{x } n_2 \text{ x } \dots \text{ x } n_m = n^m \qquad (3\text{-}2)$$

where *n* and *m* are the number of rows and columns, that is number_of_rows$^{\text{number_of_columns}}$

A bin packing algorithm solves the problem of allocating containers (bins) to items (often shipping items) in a way that attempts to use the fewest possible number of bins by assigning multiple items to a single bin where possible based on some criterion such as weight (mass) or volume and potentially value or cost of each item and/or bin. In the ECU/features problem, if solved by a bin-packing algorithm, bins equate to ECUs and items equate to features.

3.2 Exhaustive Overkill

This produces every possible combination and permutation of a binary valued vector or bit-string that represents a square matrix, for example [001011111] = $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ reading across each row starting from the top left, mapping possible pairings of features and ECUs. Unfortunately, without constraints or rules to determine which of these maps to a feasible solution (for example only allowing a single non-zero entry in any column of the *n* x *n* array) some of the string representations will always produce invalid solutions, even though mathematically possible in the context of memory allocation and cost of the ECUs with respect to the string or array that was generated.

The default position for a trivial solution is to assign a single ECU/feature pairing for every feature or to assign just one ECU to support all features. These are starting positions for the exhaustive overkill method. For a system of three features, a 3x3 array maps ECUs (rows) to features (columns) and the default positions generate the following arrays

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}, C = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}, D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

representing three features on the first ECU (A), three on the second (B), three on the third (C) or one feature on each (D). Every other 0-1, 3x3, matrix is generated by a binary counting algorithm and tested for fitness to see if it is better than any of the four solutions given by the above. There are 2^9 possible matrices, including the zero matrix of which some simply cannot be implemented as they place some feature(s) on more than one ECU or they exclude some features from being supported at all.

3.3 ESP with sufficient iterations to cover the entire feasible solution space

This method produces a single array for each and every combination of feasible solutions. The creation of each new array is concerned only with whether the mapping would be physically possible, focusing only on ensuring that the array avoids pairing any single feature with more than one ECU. Whilst this, exhaustive, method is very time consuming it reduces the search space from 2^{n^2} down to n^n. Table 3-1 shows the number of possible solutions for the Exhaustive Overkill and ESP methods for the same problems

Table 3-1:Total possible solutions for exhaustive overkill and ESP with *n x n* matrices

n	Overkill = 2^{n^2}	ESP = n^n
1	2	1
2	16	4
3	512	27
4	65536	256
5	33554432	3125
6	68719476736	46656
7	562949953421312	823543
8	1.84467E+19	16777216
9	2.41785E+24	387420489
10	1.26765E+30	10000000000
11	2.65846E+36	285311670611
12	2.23007E+43	8916100448256
13	7.48289E+50	302875106592253
14	1.00434E+59	1.1112E+16

By inspection, the values for the exhaustive overkill method beyond seven features are clearly unmanageable, whilst the values for the ESP do not become a problem before *n*=14

For any given size of ECU/Features problem, there is a finite search space which produces an optimal solution with a specific number of ECUs supporting the features. Of the matrices that can be created by any size of ECU/Features array, only a subset is feasible in terms of mapping single ECUs to one or more features in the problem. Matrices that would cause a feature to be supported on more than one ECU are infeasible in this respect. The diagram in Figure 3-4 shows all of the feasible matrices for the 4x4 problem, which are just 256 out of the 65536 different 0-1 matrices that can be generated with 16 elements. This is a reduction of the search space to 1/256 of the exhaustive search but not all reductions are to the square root of the original size. The four solutions with one row each (four features supported on just one ECU) account for 0.015625 or of all the feasible solutions, suggesting that one or other of these might be found with a probability of 1/64

There are 144 possible solutions that use three ECUs (9/16) and 84 possible ways to find a solution with only two ECUs (21/64). The 24 ways of arranging a single ECU to support each of the four features (a four ECU solution) has a probability of 3/32

Figure 3-4 shows the 256 unique mappings of the 4x4 problem with the non-zero entries highlighted to demonstrate the number of non-zero rows used in each solution.

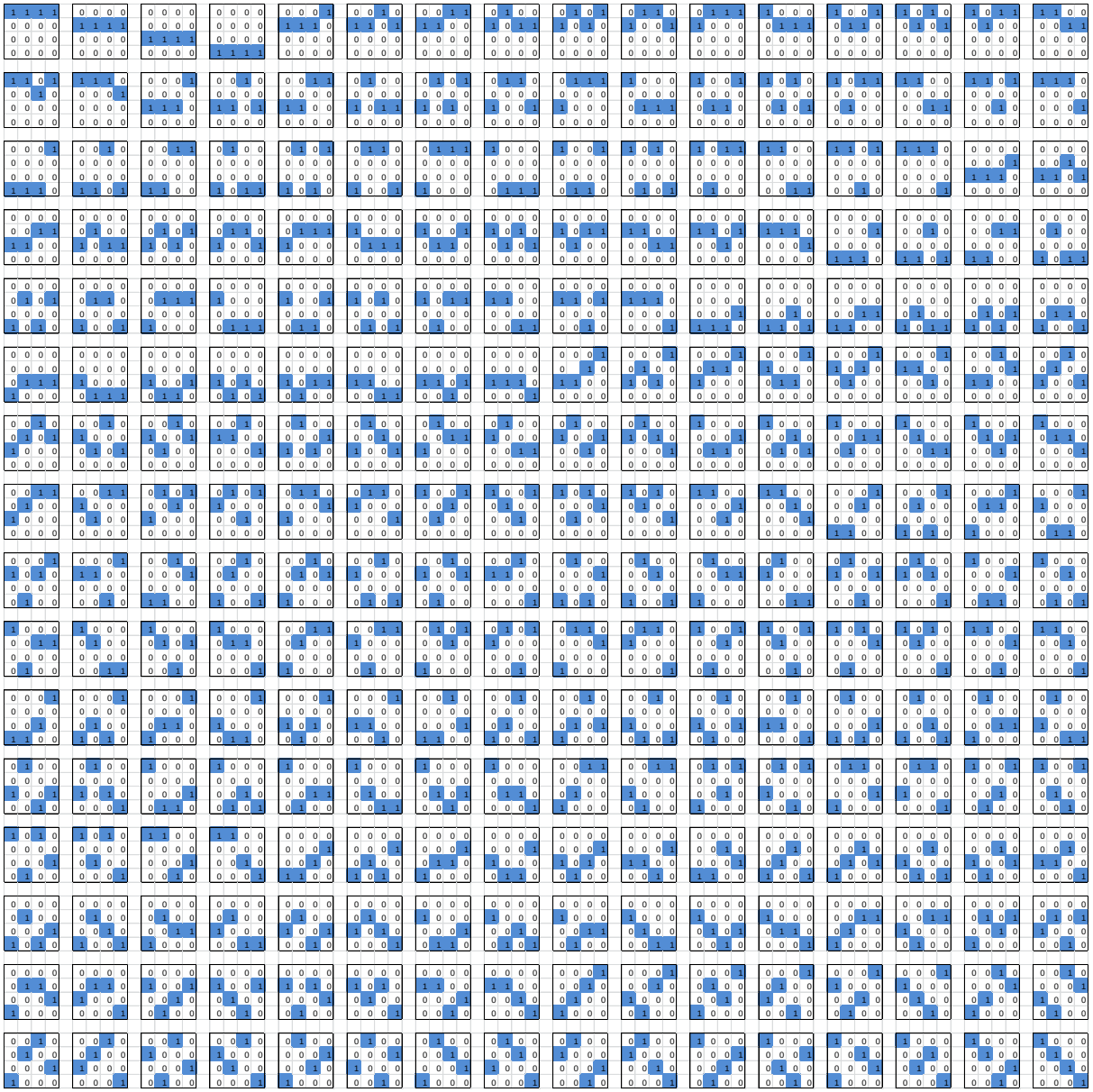


Figure 3-4: The 256 possible arrays mapping a 4 feature problem to a solution from 4 ECUs

For any problem size, the maximum possible number of feasible matrices in an *n* x *n* array is n^n and each solution will use between 1 and *n* ECUs. The solution search space for each problem has a finite number of solutions that use exactly 1 ECU and this is a function *f(n)=n* and a finite number of solutions that use *n* ECUs (one ECU for each feature and exactly one feature on each ECU) equal to *n*!

In between solutions with 1 and *n* ECUs, matrices that have non-zero entries on some number of rows between 1 and *n* exist in a finite number of ways that is also a function *f(n, r)* where *r* indicates the number of rows populated in the *n* x *n* matrix.

The function *f(n, r)* itself is less easy to calculate algebraically than it is to generate by numerical and computational methods. An algorithm for the implementation of the function

in ‘C’ is included in Appendix A. Like the values for the number of total solutions for any given size of ECU problem, values for the subset of number of rows used grow very quickly and are soon difficult to handle in 32 bit integer memory.

This means that, for practical purposes, only the first ten arrays (1x1 to 10x10) could be attempted in this research but the values obtained give an insight into the probability of a solution being offered that uses any number of ECUs if the solution were chosen randomly.

For example, in the trivial case of only one feature, there is a 100% probability that the solution will be realised on just one ECU.

The total number of distinct matrices for the 2x2 problem is 16, of which only four can provide feasible solutions. Two solutions use two ECUs and a further two use just 1 ECU. The probability of any of these being chosen is 2/4 or 0.5 or 50%.

The chart in Figure 3-5 shows the probabilities associated with any solution offered for a given number of features in the problem. This is used to validate the GA when solutions are offered in more than the expected percentage of solutions over a large number of runs.

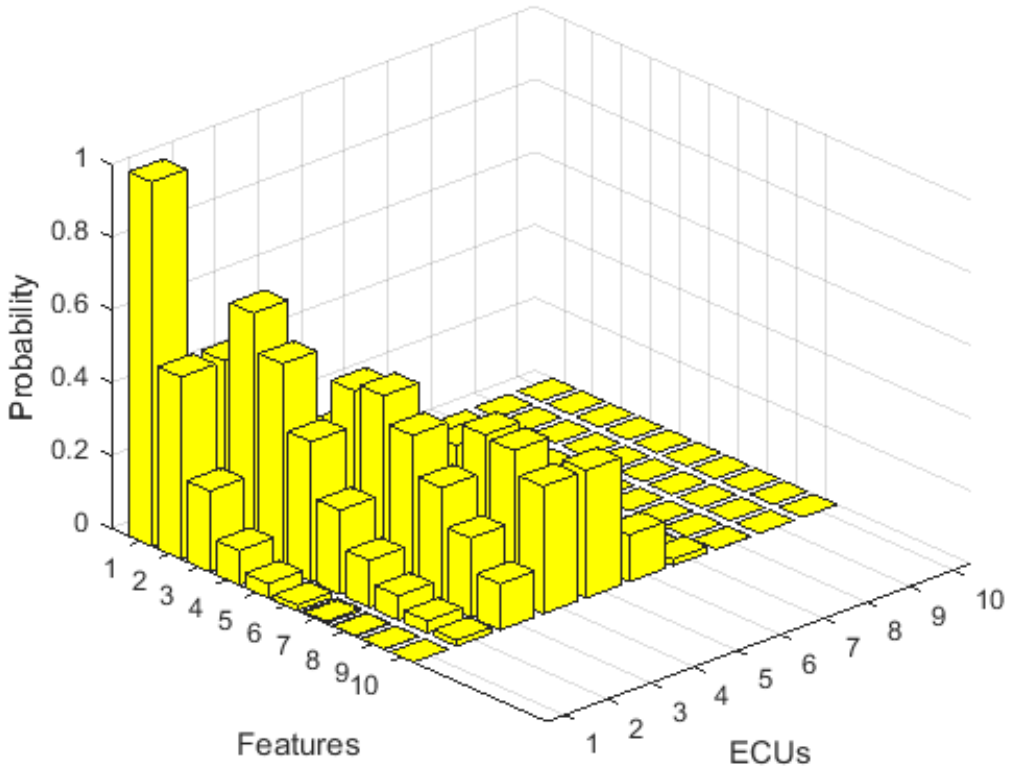


Figure 3-5: chart of the probabilities of any ECU/Features combinations chosen randomly

In tabular form the values from figure 3-5 are those in table 3-2, showing probabilities associated with supporting a given number of features (row data) on any number of ECUs (column data) so that, for example, the probability that a six feature system will find a solution on two ECUs is 23.15%.

Table 3-2: Probabilities of ECU usage by numbers of features

		ECUs										
		1	2	3	4	5	6	7	8	9	10	Total Prob
Features	1	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.00000
	2	0.5000	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.00000
	3	0.2222	0.6667	0.1111	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.00000
	4	0.0938	0.5625	0.3281	0.0156	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.00000
	5	0.0384	0.3840	0.4800	0.0960	0.0016	0.0000	0.0000	0.0000	0.0000	0.0000	1.00000
	6	0.0154	0.2315	0.5015	0.2315	0.0199	0.0001	0.0000	0.0000	0.0000	0.0000	1.00000
	7	0.0061	0.1285	0.4284	0.3570	0.0768	0.0032	0.0000	0.0000	0.0000	0.0000	1.00000
	8	0.0024	0.0673	0.3196	0.4206	0.1703	0.0193	0.0004	0.0000	0.0000	0.0000	1.00000
	9	0.0009	0.0337	0.2164	0.4131	0.2713	0.0606	0.0039	0.0000	0.0000	0.0000	1.00000
	10	0.0004	0.0163	0.1261	0.35	0.35	0.1286	0.0172	0.0007	0.0000	0.0000	0.98923

The greyed out values in table 3-2 are values that could not be calculated within the limits of 32 bit arithmetic on a desktop PC and have been estimated by giving them both equal values that complement the sum of the probabilities to 1.

Revisions to the number of variables in the ECU problem are possible to include, for example memory capacity, processing speed, communication speed, cost and weight of ECUs assigned to support the requirements of vehicle hardware/software features and the multi-variable model adopted in this research is concerned with ROM, processor speed, communication speed and cost-to-purchase.

3.4 Optimisation by Bin-Packing heuristics algorithms

The allocation of ECU resources to system features is equivalent to a bin-packing problem where the ECUs are the bins or containers and the memory capacity of the ECU is equivalent to the volume or the maximum allowable mass of whatever is being put into the bins. The features are objects that need to be stored in the bins and the requirements for memory, processing speed, speed of communication etc., are catered for or constrained by the limitations of the ECUs.

A heuristic bin-packing algorithm, such as ‘Best First’ can successfully allocate ECUs to one or more features in far less time by considering the features in size order and attempting to allocate them in turn to the ECU with the largest remaining capacity, before moving on to the next feature.

When more than one variable is being considered in the bin packing algorithm, the choice about which variable determines the order of either the bins or the data (the ECUs or the features) depends on the units and scale of measure being used. It’s hard to compare memory capacity with the price of a specific ECU, other than determining a general correlation between the two and presuming this holds in all cases. In the case of even the same units measuring memory capacity of program storage space and dynamic memory for global variables, the scale of the two can differ so much that it is meaningless to compare for example 32256 bytes of one against 2048 bytes of the other. Worse still, determining which ECU has greater overall capacity based on as few as these two variables, how does (for example) 31000 bytes of program storage space and 1996 bytes of dynamic memory compare with, say, 32024 and 1024 bytes respectively? Should they be summed, averaged, normalised?

By taking the square root of the sum of the squares of the two variables, a value can be calculated as a function of the two variables and these values are plotted for $0 \leq X, Y \leq 10$ in Figure 3-6 where heights of the bars show the values of $f(x, y) = \sqrt{(x^2 + y^2)}$

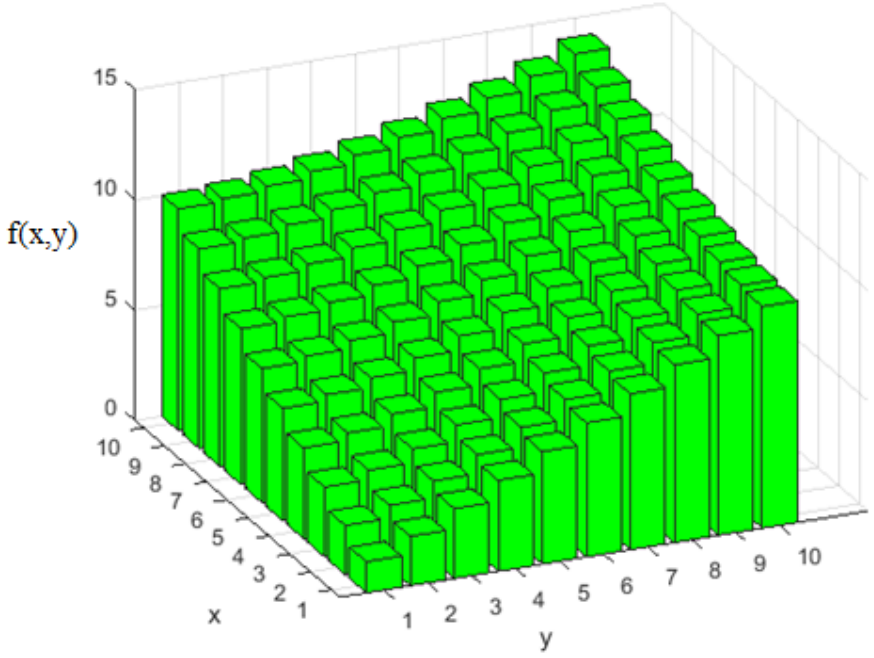


Figure 3-6: Distribution of square roots of sums of squares.

Clearly, by inspection, the value of the function for {x=10, y=10} is the greatest and the value for {x=1, y=1} is the smallest but the relative values and the order of other values is not as obvious. Why, for example, should *f*(1,9) or *f*(9,1) be greater than *f*(5,5) in terms of whether it is more desirable to have equal and mid-sized capacities in two variables or to have a very large capacity of one and very little of another?

Less easy to visualise for the 3x3 problem, this is like being given a number of each of three differently coloured counters or gambling chips with no quantitative values ascribed to any of them. Does a pile of 10 reds, three blues and six greens come before or after a pile of seven reds, six blues and six greens? The total is 19 chips in both scenarios but is it better to have a large number of one or to have roughly equal numbers of all? This is like having more memory and less processing speed or having an average amount of both.

One solution is simply to take the arithmetic mean of the different properties of the ECU (for example, all types of memory, processor speed and price) and scale them so as to have similar ranges before applying an averaging or normalising process.

Another method is to take the square root of the sum of squares across the different properties. Applied to this problem, it gives values for any number of variables that equates to a geometric interpretation of the values as sides of an *n*-dimensional right angled figure. In the two-variable case, this is the hypotenuse of a right-angled triangle and, in the three-variable

case, it is the diagonal of a cuboid between the two furthest corners. As more variables are introduced, the value obtained becomes a vector in *n*-space that represents the hypotenuse of an n-dimensional right-angled shape.

For two variables, each with maximum values of for example 10, it should be clear that the worst possible combination of the two is [0,0] and the best is [10,10] and somewhere near the middle is [5,5] but how would for example [3,7] compare with [6,4]?

Table 3-3 : Results of taking the square root of sums of squares of column headings and row headings

	1	2	3	4	5	6	7	8	9	10
1	1.41	2.24	3.16	4.12	5.10	6.08	7.07	8.06	9.06	10.05
2	2.24	2.83	3.61	4.47	5.39	6.32	7.28	8.25	9.22	10.20
3	3.16	3.61	4.24	5.00	5.83	6.71	7.62	8.54	9.49	10.44
4	4.12	4.47	5.00	5.66	6.40	7.21	8.06	8.94	9.85	10.77
5	5.10	5.39	5.83	6.40	7.07	7.81	8.60	9.43	10.30	11.18
6	6.08	6.32	6.71	7.21	7.81	8.49	9.22	10.00	10.82	11.66
7	7.07	7.28	7.62	8.06	8.60	9.22	9.90	10.63	11.40	12.21
8	8.06	8.25	8.54	8.94	9.43	10.00	10.63	11.31	12.04	12.81
9	9.06	9.22	9.49	9.85	10.30	10.82	11.40	12.04	12.73	13.45
10	10.05	10.20	10.44	10.77	11.18	11.66	12.21	12.81	13.45	14.14

Taking the root of sum of squares of 3 and 7, we get $\sqrt{3^2 + 7^2} = \sqrt{58}$ whilst 6 and 4 gives $\sqrt{6^2 + 4^2} = \sqrt{52}$ meaning that 3 and 7 score better together than 6 and 4. The pairing of 5 and 5 gives $\sqrt{50}$ which has the lowest score of any pair that sum to ten.

To see the effects of three interacting variables, each with values between 1 and 5 inclusive, the composite values of the three are given denary values with each variable taking a different place column, that is a=1, b=2, c=3 which would be represented by the three figure number in base 10 equal to 123, just to give it somewhere to appear on a graph. Its actual order would be determined by the value $\sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$ which would place it higher than a=2, b=2, c=2 $= \sqrt{12}$ which becomes difficult to show in two dimensions. It is easy to show the relatively larger or smaller values of each triple as the heights of a bar chart or graph plotted against the composite triple, that is if the x-axis shows the triple and the y-axis shows the root of sums of squares, a neat and tidy line graph or bar chart can be drawn where the square root values are not in order but the composite triples are.

Turning this around so that the root of sums of squares values are in order and the triples can be read from the y-axis, the order but not the scale is preserved for the square root values which now become the data labels on the x axis and the heights of bars or the positions of points on a line graph are the triples which can be read as three digit base ten numbers.

When the order of the ECUs (bins) is changed before compilation and execution, there is no change in the result because the bins are ordered during program execution and before any fitting of items into bins is attempted.

However, if the order of the features (items) is changed, this fundamentally alters the algorithm that attempts to fit the features to the ECUs (bins) because no ordering is carried out on the items after compilation. This was beyond the scope of this research and could be an area for further work in the future.

Therefore, the manual ordering of items can have a lesser or greater effect on the ability to fit items because the 'non-changing' algorithm will attempt to fit smaller items to larger bins that can then no longer be used for large items that come along later.

The generally accepted method for this type of bin packing ("Best fit, Largest First" or "First Fit") described by Martello and Toth [59] is to order the bins (largest to smallest) and order the items (also largest to smallest) before attempting any fitting (largest to largest available, already used if possible). This is borne out in the results of scenarios 18 and 19 in section 3.5

Since the ESP increments the binary value of the vector representing the matrix for the next fitness test by a deterministic algorithm, the order of the features and ECUs will have an effect on the output because configurations (architectures) that have the same number of ECUs will be examined in a different order if their positions in the ECUs/Features array is altered and, because the fitness function is ultimately concerned with the number of ECUs that are used, finding two equally suitable architectures will result in one being chosen over the other and vice versa if that specific solution was visited first by the ESP.

Only a better solution will overwrite a currently stored best solution. Equally good solutions will be discarded if they are found later in the execution. This is why a fitness score that takes the value of each ECU used would be helpful in predicting a more deterministic result – or at least one that is unique.

3.4.1 First Fit Algorithm

Consider the requirements of the features of the system. In the case of the ECU/features problem, the simplest property of the ECU and feature is memory. If the features are ordered by their required memory for each feature, largest to smallest and if the ECUs are ordered by their memory capacity, also largest to smallest, the features and ECUs can be matched starting with the largest pair (the first ECU and the first feature).

Since it should be possible to place every feature on to separate ECUs, every ECU will have been chosen in advance to be able to support at least one of the features at the same position in the list. The values for the features' memory requirements in Table 3-4 are purely arbitrary and the ECU memory sizes are representative of possible ECU configurations with the memory size being a power of two as would be the case in commercially available ECUs. The actual values chosen for these ECU memory capacities are either the nearest power of

two above the corresponding feature memory requirement or a value that would allow more than one feature to be supported.

Table 3-4: Example of ECUs and ten features paired by a bin-packing heuristic algorithm

ID number	1	2	3	4	5	6	7	8	9	10	Total
ECU Memory Size	4096	4096	4096	2048	2048	1024	512	512	512	512	19456
Feature Memory Requirement	3042	2048	2048	2048	1024	512	512	512	512	512	12770
Feature	1	2+3	4+5+6+7	8+9+10	Unused	Unused	Unused	Unused	Unused	Unused	6686
Remaining capcity	1054	0	0	512	2048	1024	512	512	512	512	6686

As well as the 1:1 solution to the data in figure 3-3 one possible solution using fewer ECUs but still supporting all of the features is shown here, the result having been derived from a bin-packing algorithm discussed in detail later in this thesis. The suggested solution based on bin packing is as follows

ECU : 1 supports : 1 : 19 : 20
ECU : 2 supports : 2
ECU : 3 supports : 3
ECU : 4 supports : 4
ECU : 5 supports : 5
ECU : 6 supports : 6
ECU : 7 supports : 7
ECU : 8 supports : 8
ECU : 9 supports : 9
ECU : 10 supports : 10 : 16
ECU : 11 supports : 11 : 14
ECU : 12 supports : 12 : 15
ECU : 13 supports : 13 : 17
ECU : 14 supports : 18

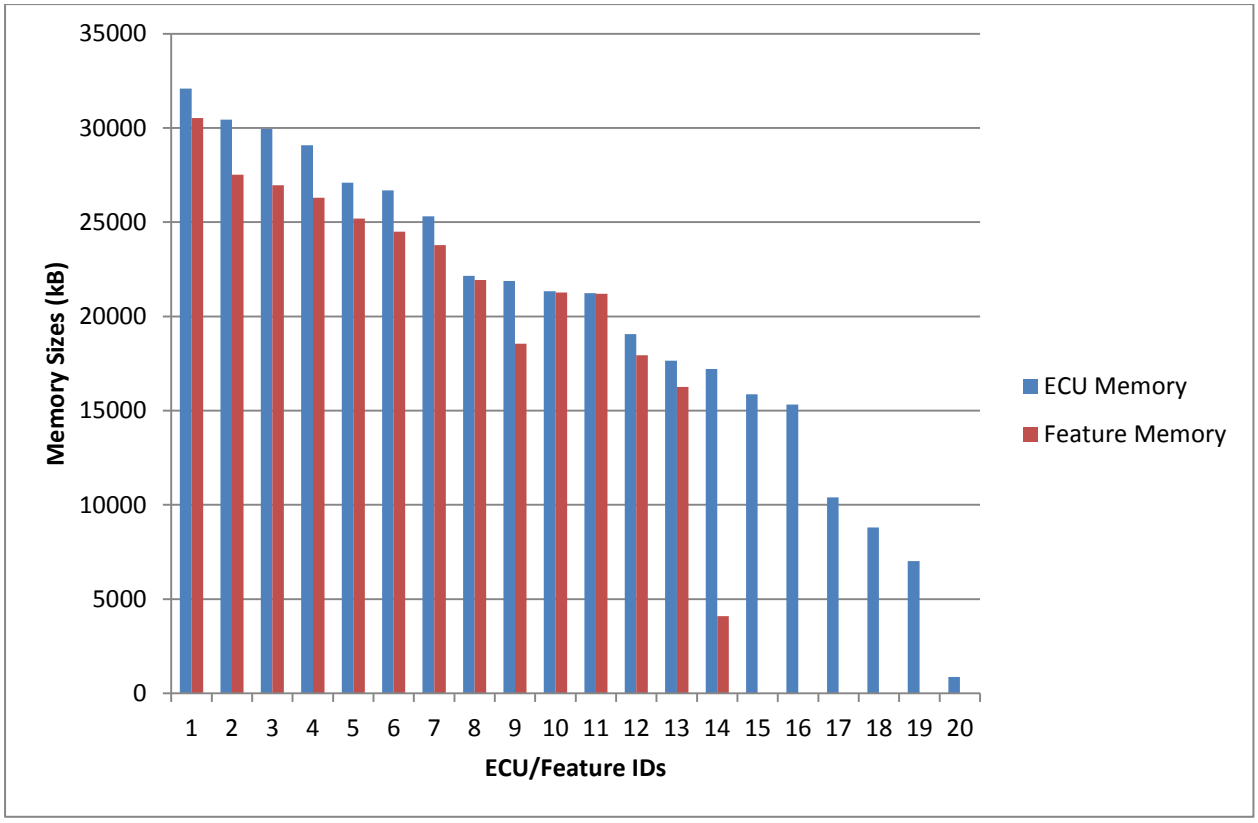


Figure 3-7 : Candidate solution from Bin Packing algorithm showing total feature memory allocated to each ECU

Because the data are either presented in order, largest to smallest, or because they can be sorted from the outset by the bin-packing code, a 'first-fit' or a 'first-fit-decreasing' algorithm is used to allocate items to bins (in this specific case, allocating features to ECUs).

The number of steps needed to perform a best first bin packing algorithm where the number of bins is equal to the number of items is between n and n(n+1)/2 where n is the number of bins and is equal to the number of items.

The best case, in which the least iterations are required, is where all of the items will fit into the largest bin although this could be wasteful of spare capacity if that bin is excessively large. The bin packing algorithm has no way of checking for this and so without further modification of the algorithm to include comparisons of bin sizes, this is a possible scenario that might be sub-optimal. Figure 3-8 shows a scenario where all item fit into the first bin (all features are supported on the one ECU). Figure 3-9 shows the maximum possible number of steps required when each bin in turn can fit only one item but with some spare capacity that needs to be tested against the size of the next item every time a new item is selected. The result of this scenario is that the number of steps taken is the triangular number for *n*=number of bins.

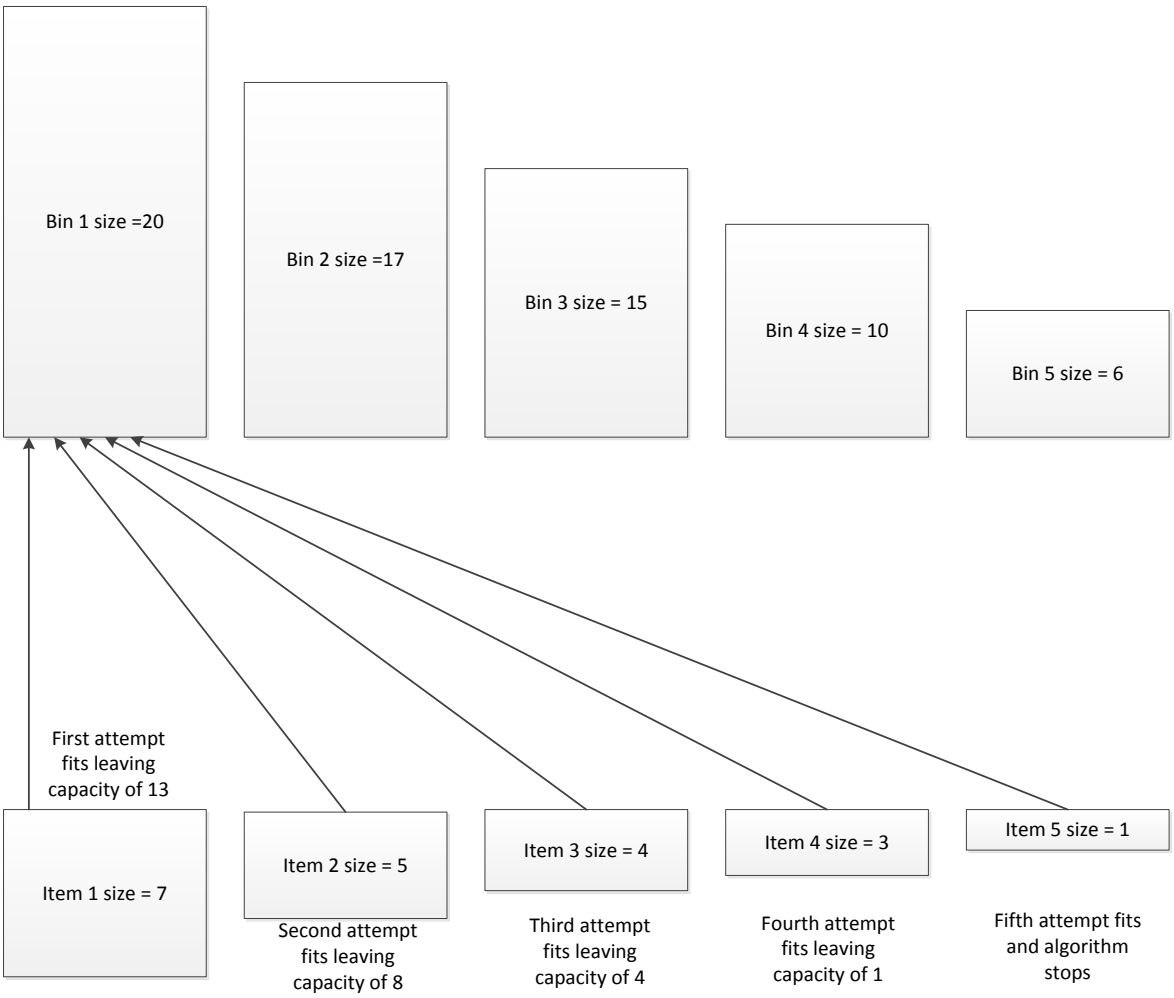


Figure 3-8: Bin packing problem using minimum possible steps = *n*, where number of bins and items are both equal to *n*

Because each item needs as many tests as its position number in the queue, the number of steps is the sum of the attempts, 1+2+3+4+5….. and this generates the sequence of triangular numbers, 1, 3, 6, 10, 15, 21….. for as many items as are in the queue. This can be generated by the function $y=f(n) = n(n+1)/2$ so that for the 100 features problem, the bin packing algorithm only requires 5050 steps.

Table 3-5: Comparison of best and worst possible performances of GA and Bin Packing compared with ESP

		GA			Bin packing	
n	ESP	Best	Usual	Worst	Best	Worst
1	1	1	1	1	1	1
2	4	2	2	4	2	3
3	27	3	8	16	3	6
4	256	4	<64	256	4	10
5	3125	5	<128	2048	5	15
6	46656	6	<512	32768	6	21

When each item fits each bin exactly with no spare capacity, as in figure 3-9, or when every bin is filled exactly by one or more items before the next bin has been used, the algorithm uses the minimum number of steps.

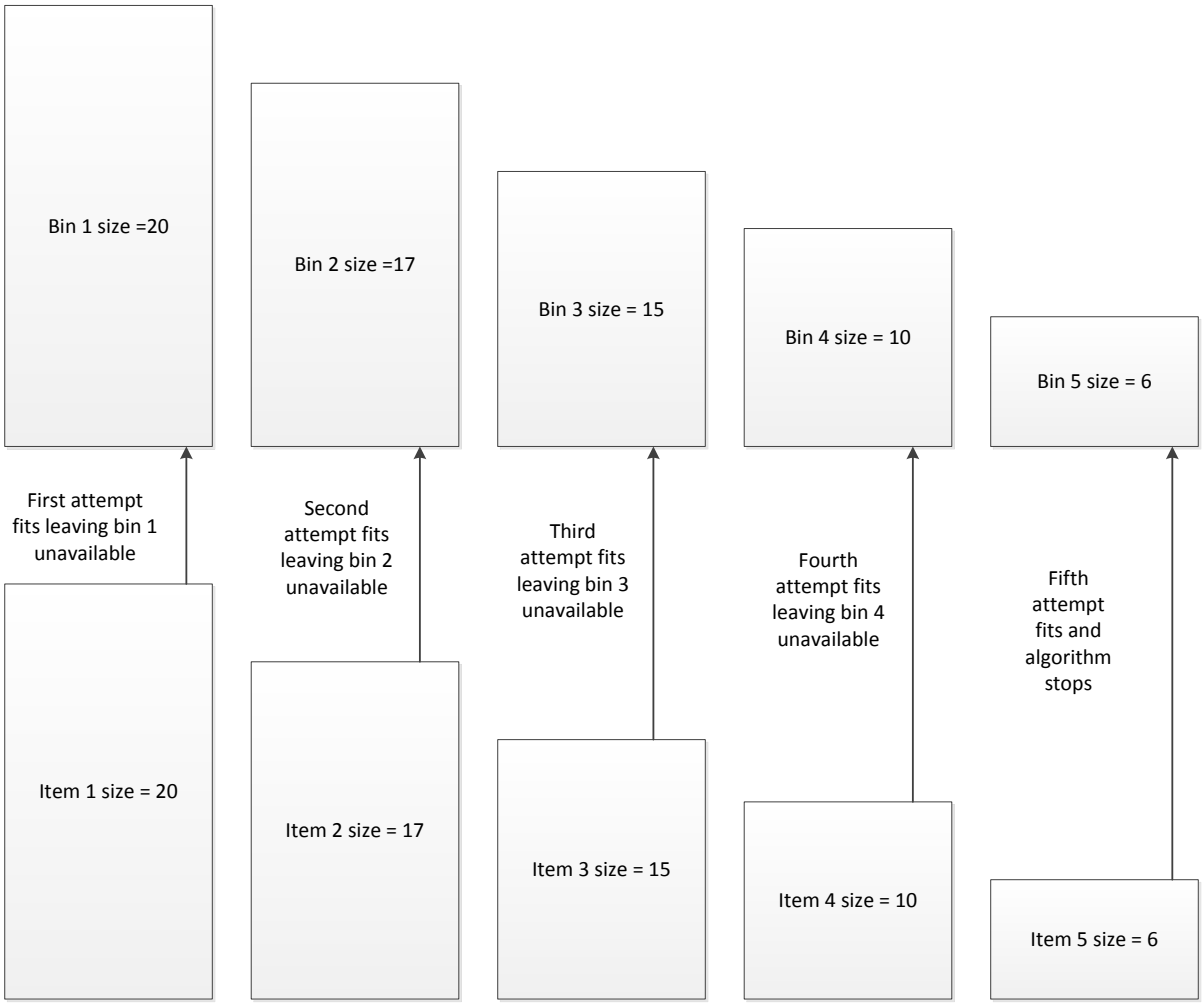


Figure 3-9: Example bin packing scenario where every bin is filled completely before the next bin has been tested or used

Figure 3-10 shows the more complex sequence of events that lead to the worst case scenario where there is some spare capacity on every bin even after it has been used which causes the algorithm to explore every possible bin as a candidate for each item up to the first bin into which it will fit.

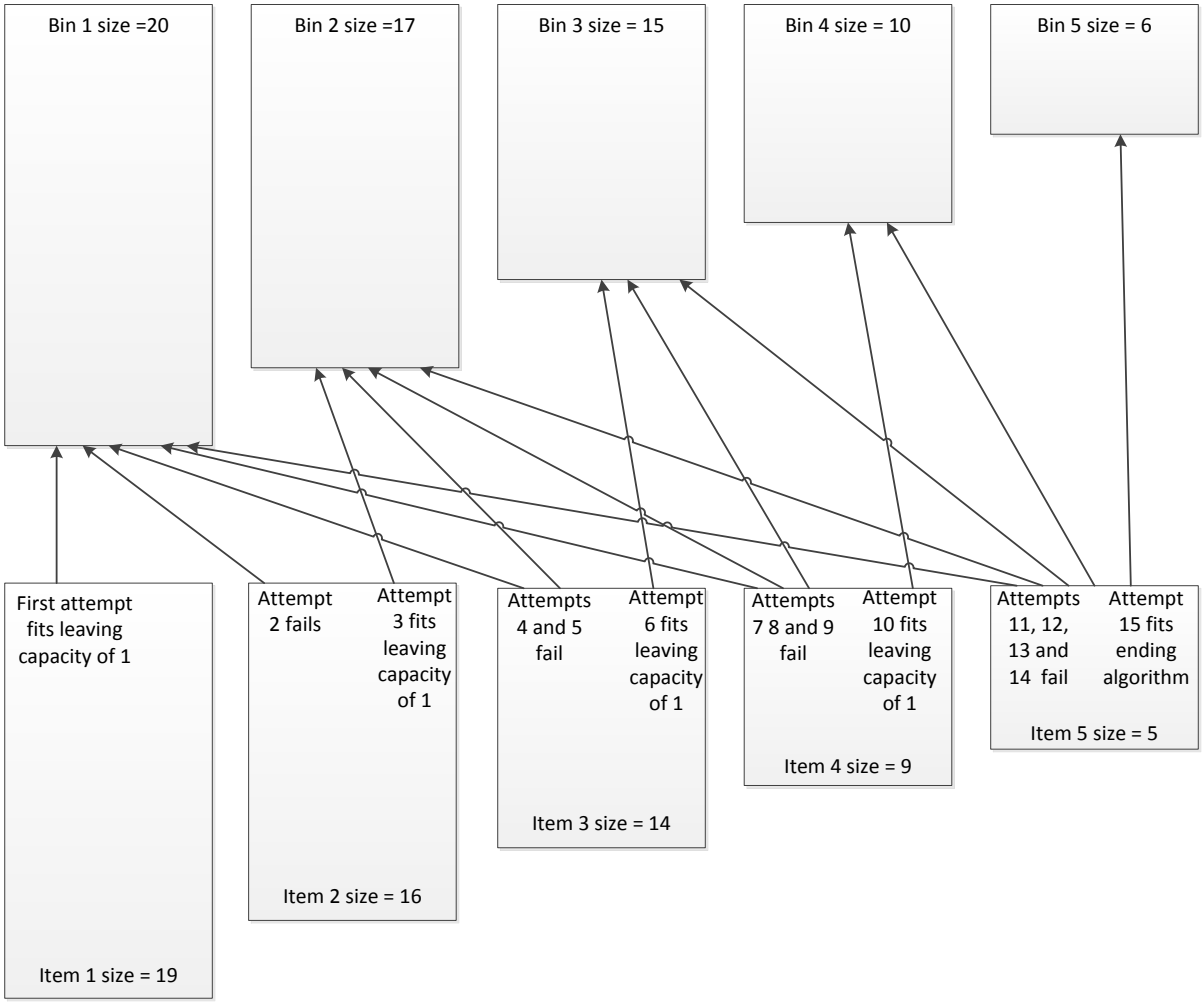


Figure 3-10: The worst case of the bin packing scenarios in which items almost fill each bin, leaving it available to be checked at the next selection

3.4.2 Multi-variable model

For a system of greater complexity, with more than one variable, the bin-packing algorithm needs to sort the bins and items (ECUs and features) on some combination of the variables that allows a value based on every variable to be used as the sorting value. For example, in the system that uses program storage space, local variable storage and processor-speed as the features' requirements that need to be satisfied by the ECU's capacity in these three variables, how should one sort the ECUs?

For example, it is useful to consider a hypothetical model for the moment that will be introduced as a problem to be solved by this research in a later chapter, to be implemented on one or more ECUs The values attributed to the features are, in no particular order, as in Table 3-6 and are sorted on the value of a calculated "n-dimensional hypotenuse" at runtime.

Table 3-6 : ECU requirements for 13 feature problem

ID	Program Space	Local Variables	Processor Speed	n-dimensional Hypotenuse
1	14692	551	4000	15236.747
2	7576	277	4000	8571.610
3	7264	245	4000	8296.127
4	4314	222	4000	5887.264
5	6294	339	4000	7465.210
6	7644	293	4000	8632.299
7	5482	226	4000	6789.948
8	5354	364	4000	6693.117
9	5298	223	4000	6642.178
10	4852	214	4000	6291.876
11	8910	305	4000	9771.444
12	6306	289	4000	7473.229
13	5676	259	4000	6948.673

The *n*-dimensional hypotenuse in table 3-6 is the square root of the sums of squares calculated on the three variables for each individual feature, for example, for the variables associated with feature number 1, whose values are 14692, 551 & 4000, the value which this research will refer to as an *n*-dimensional hypotenuse is the magnitude of a 3-D vector calculated as the square root of sum of squares

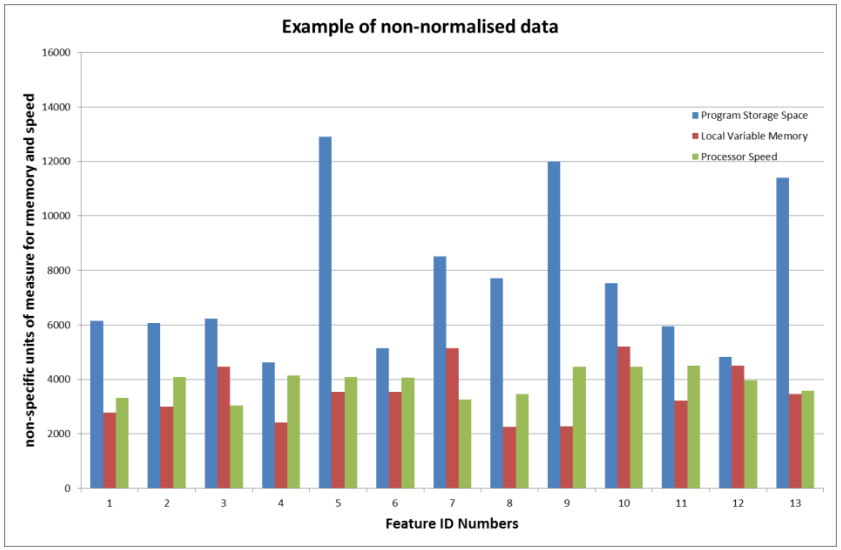
$$\sqrt{14692^2 + 551^2 + 4000^2} = 15236.7472$$

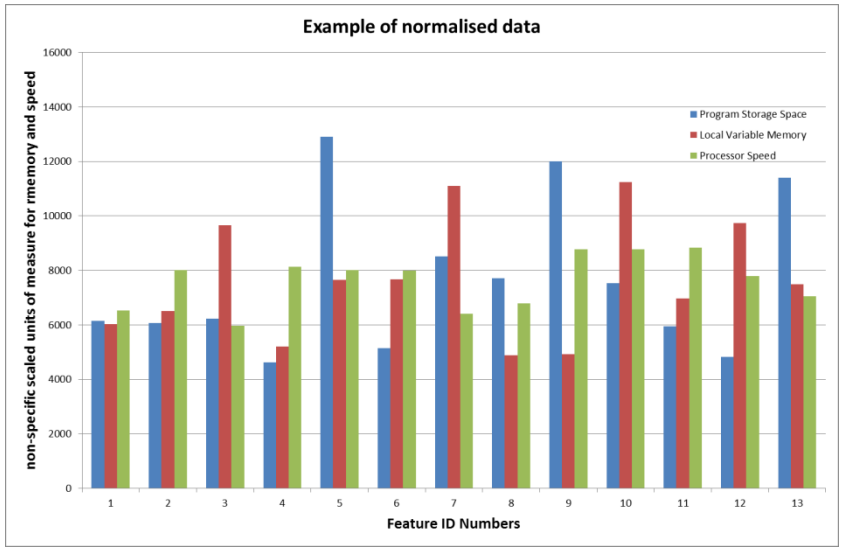
For greater than three dimensions, this is extended to the magnitude of an n-dimensional vector. A refinement to this is to scale the values of each variable so that the difference in units of measure and therefore in absolute size of each variable does not unreasonably affect dominance of one variable over the others, that is if the measures of program storage space are always in the range of 4,000 to 15,000 for example whilst the measure of local variable memory are always less than 600, the value of the program storage space will dominate the hypotenuse calculated and have greater influence on the sorted order of the features and ECUs.

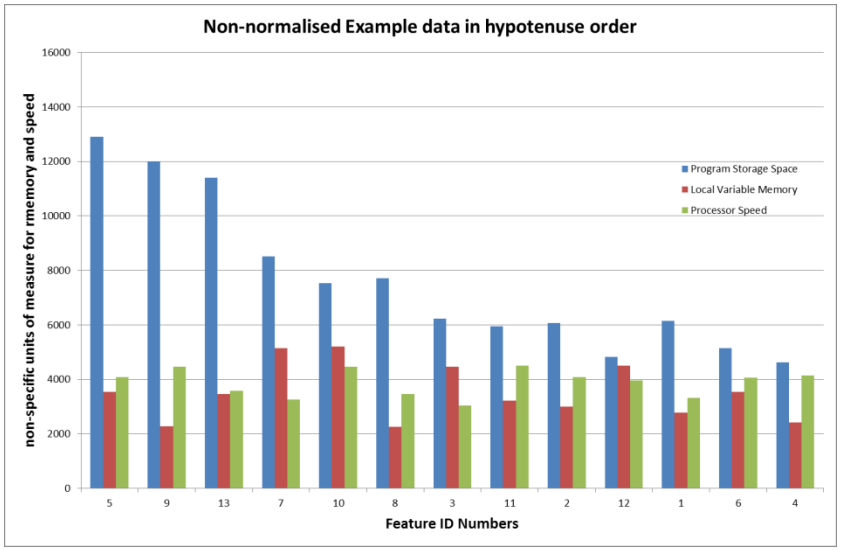
By statistically adjusting the size of each variable to maintain their relative sizes and make their individual spread characteristics similar across all groups (storage space values remain relatively unchanged within the group of ‘storage space’ and local variable values remain relatively unchanged within the group of ‘local variable’) the influence of one variable on the sorted order is diminished.

Scaling is performed according to the average value of each group relative to the largest (program storage space) values. Each individual value of the other variables is based on a scaling factor of average(V_1)/average(V_x)

The four graphs in Figure 3-11, show the effects of the normalisation process employed by the bin-packing algorithm to remove bias towards one variable (usually program storage space). A randomly generated and feasible set of requirements for an example of 13 separate features has been normalised and then ranked according to a square root of sum of squares calculation. Calculation of the square root of sum of squares would also provide a scalar value that is a combination of the three variables suitable for ordering.







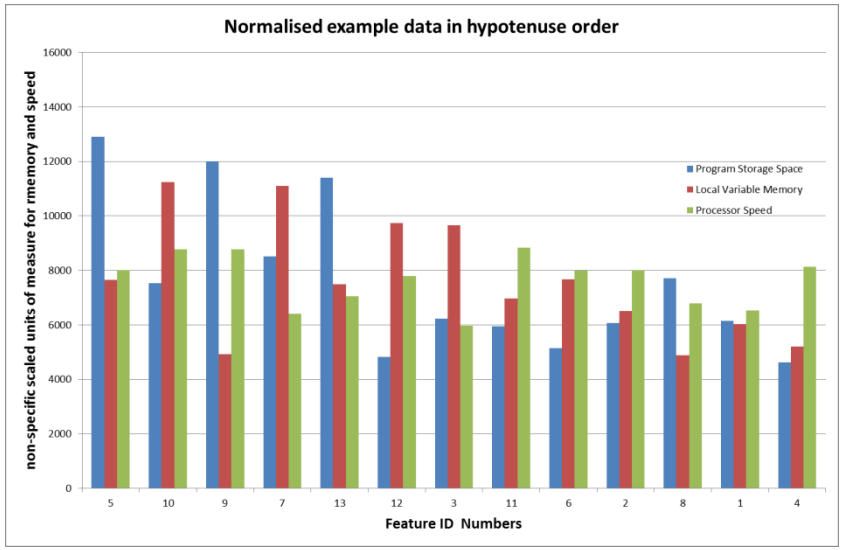


Figure 3-11 : Normalised and non-normalised data before and after sorting

In the case of non-normalised data, the units of measure and quantities of these for very different variables (for example program storage space, processor speed, communication speed etc.,) program storage space (blue) dominates and ordering based on this variable alone would produce the same result as sorting on a calculated hypotenuse, with only one exception (ID 12) where the non-normalised variables are already very close in value initially. With the normalised data, no single variable dominates and the order is less biased towards the program storage space variable.

3.5 Genetic Algorithm

Genetic algorithms (GAs) attributed to John Holland in [62] belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution. They are particularly suited to problems with continuous 2D and 3D search spaces and known or knowable generating functions that are differentiable for solutions across the entire domain.

Genetic algorithms are normally initialised by a randomly or arbitrarily selected starting point where the solution is not known (until generated by the generating function) or to start with a known solution and its input value(s). Following this, the properties of the gradient at that point and points nearby inform the GA where to search next for a better solution.

If a maximisation is required, a value that is further up the slope of a graph is preferable to one that is further down the slope and following the gradient upwards will cause the GA to arrive at a local maximum.

The non-continuous, discrete, nature of the ECU problem means that it cannot be solved by derivative-based search algorithms that use the gradient of a function or curve to determine the direction of the search towards local and global maxima or minima.

For the continuous model, the GA moves up or down a slope in a landscape of mountains and valleys. This can be explained in two dimensions with the following graphs of arbitrary differentiable functions, chosen because the negative value raised to the power of combinations of sin and cos is known to produce turning points with positive and negative values for *f*(t). The addition of 8 at the end of the function raises all of the values for *f*(t) to positive values. The graph in Figure 3-12 shows a plot of the arbitrarily chosen continuous and differentiable function

$$y = f(t) = -1^{t}\left(\sin\left(\frac{t}{2}\right) + \sin\left(\frac{t}{4}\right) + \sin\left(\frac{t}{32}\right) + cos(t) - \frac{t}{8}\right) + 8\,;\ 1 \le t \le 64 \qquad (3\text{-}3)$$

and for clarity, the same function is plotted over the domain $1 \le t \le 16$ in Figure 3-13

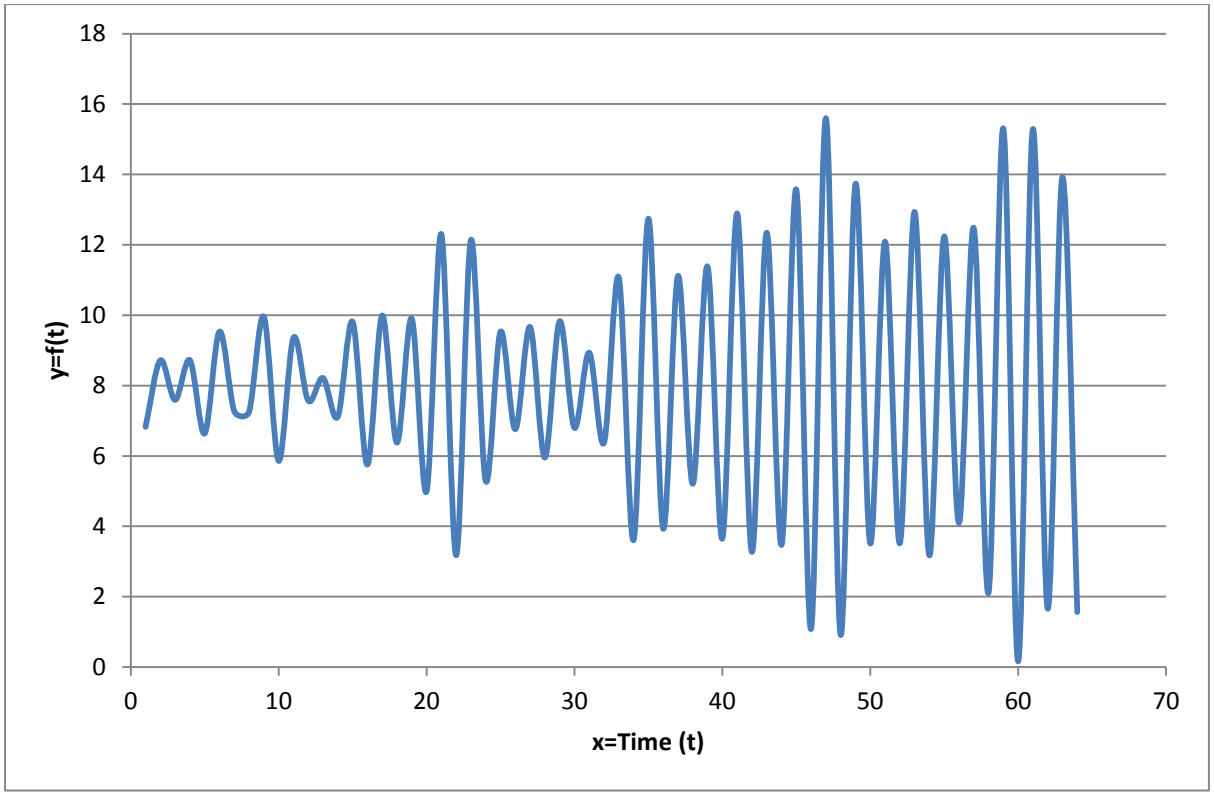


Figure 3-12 : a continuous and differentiable function y=*f(t)*; $1 \leq t \leq 64$

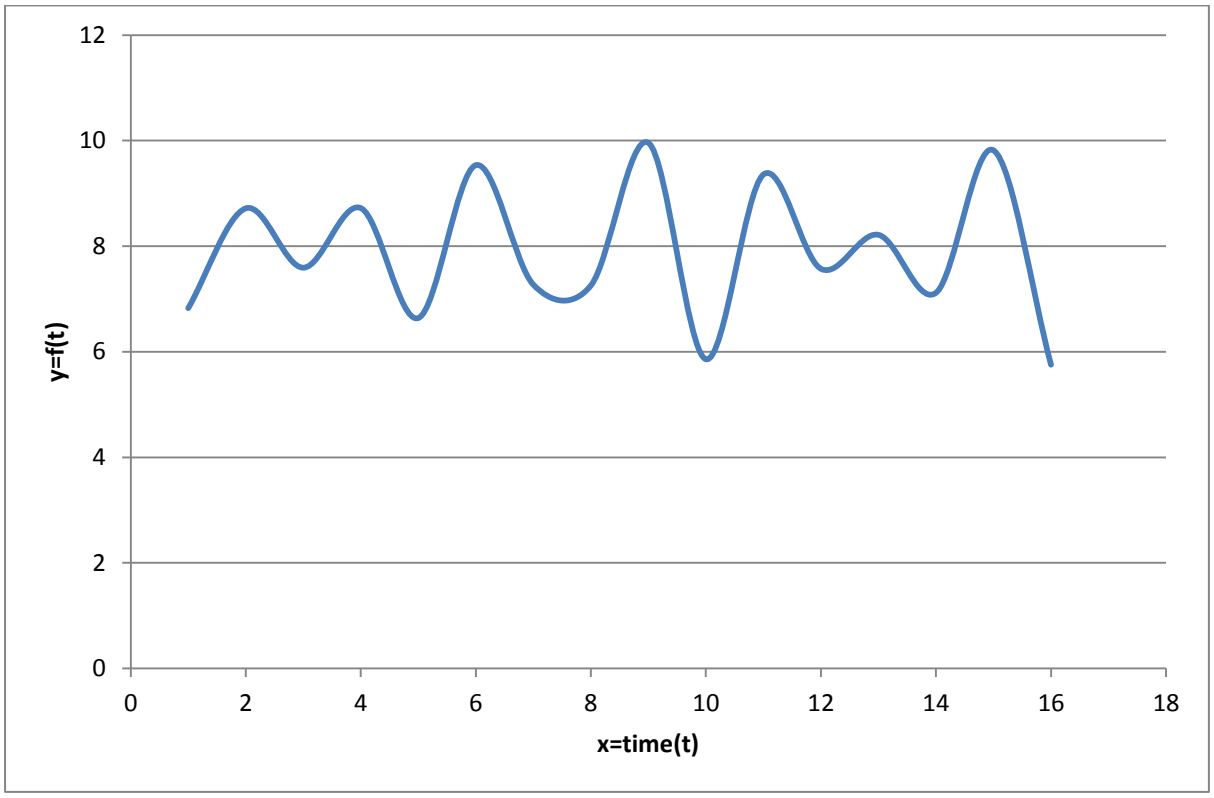


Figure 3-13 : Zooming in on the graph of y=*f(t)*; $1 \leq t \leq 16$

Since the function is continuous and differentiable, the local and global maxima/minima can be found where the first derivative is zero. Although a GA could be used to examine various points along the graph, determine their gradient and follow the path of the graph upward or downward to the nearest local maxima or minima, there seems little point if the function is already known in advance. However, if the shape of the function can be discovered by a GA and if it is known that the function will be continuous, then this method could be used to find solutions at or near turning points.

Four distinct methods are used in this research for iterating the GA and testing convergence. From the extreme of an exhaustive overkill program where every possible chromosome (even some infeasible ones) is examined and ranked according to a fitness score, discussed previously (section 3.2) via a reduced exhaustive search program (ESP) in section 3.3, to the testing of a reduced search over a problem space whose optimum solution cannot be confirmed, the GAs perform combinations of searches that either stop after a given amount of time (giving up, section 3.5.6) or which conclude when it seems likely that no more solutions will be found in a similar amount of time that has already been taken (section 3.5.5). A method for determining, geometrically, the precise locations of only the feasible chromosomes is used in the most efficient of these GA methods.

3.5.1 Chromosome example using the knapsack problem

The knapsack problem, described by Martello [59] represents items placed into a knapsack based on size or mass traded off against financial or (quantitative) sentimental values and constrained by total mass or size to achieve a maximum value within that constraint. For the purposes of this example the words ‘mass’ and ‘weight’ will be used interchangeably. This specific problem and a related family of problems serve to introduce methods for tackling the ECU problem, which can be seen as an extension to the knapsack problem with features of the vehicle having finite numerical capacity requirements that need to be supported and fulfilled by the ECUs. In this way, features can be seen as items that need to be placed into containers (ECUs).

The chromosome for a knapsack GA is an array of bits that each has only a value of either one or zero. For example, if there are ten items to choose from, the chromosome will be an array of 10 bits which represent either being placed into the knapsack (1) or not (0).

Thus, a chromosome of, say, C=[1 1 0 1 0 0 1 0 1 0] shows that five items have been selected for placing in the knapsack. Other arrays will hold the values and the weights of the objects and an objective or fitness function will sum the weights and values to determine whether the selection is within the constraints and whether it is more or less valuable than other suitable selections. That is, the mass (kg) could be represented as another 10x1 array, for example W=[5 2 3 10 7 10 5 4 18 20] and the values as another 10x1 array, for example V=[10 1 15 23 9 3 21 50 22 8]

If a constraint of 25kg total mass is placed on the selection, the chosen chromosome C=[1 1 0 1 0 0 1 0 1 0] will yield a value of 77 (unspecified units) and a total mass of 40kg. This will

be too heavy to carry according to our arbitrarily imposed maximum weight and the selection will be rejected.

The optimum selection is with a total mass of 24kg and a total value of 110 achieved with a selection of C=[0 1 1 1 0 0 1 1 0 0]

With 1024 possible selections, the genetic algorithm (code provided in Appendix B) regularly finds the optimal solution (on more than 50% of executions) after fewer than 100 iterations. The chart in Figure 3-14 shows the entire solution space to be searched for this problem. It is not obvious by optical inspection which combination of red (mass) and blue (value) yields the optimal solution. By inspection, the GA converges on the region around chromosome number 460 = [0111001100]

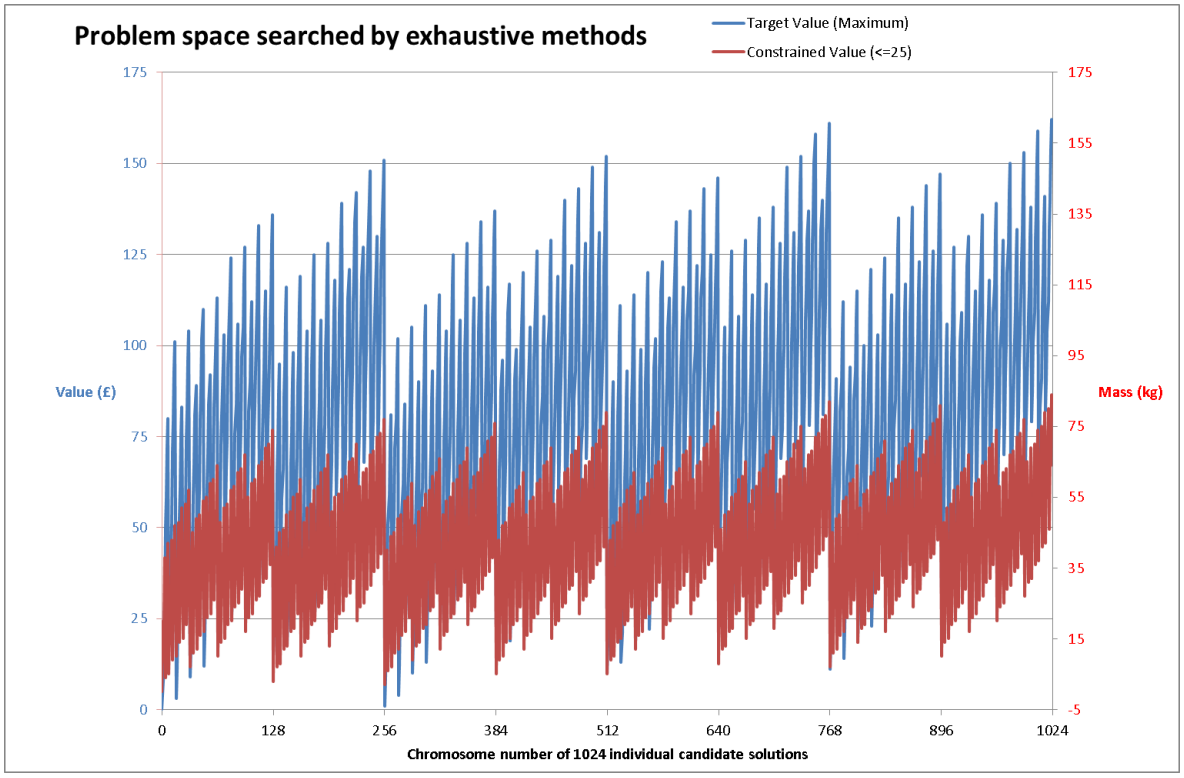


Figure 3-14: 1024 individual candidate solutions for a knapsack problem with 10 items

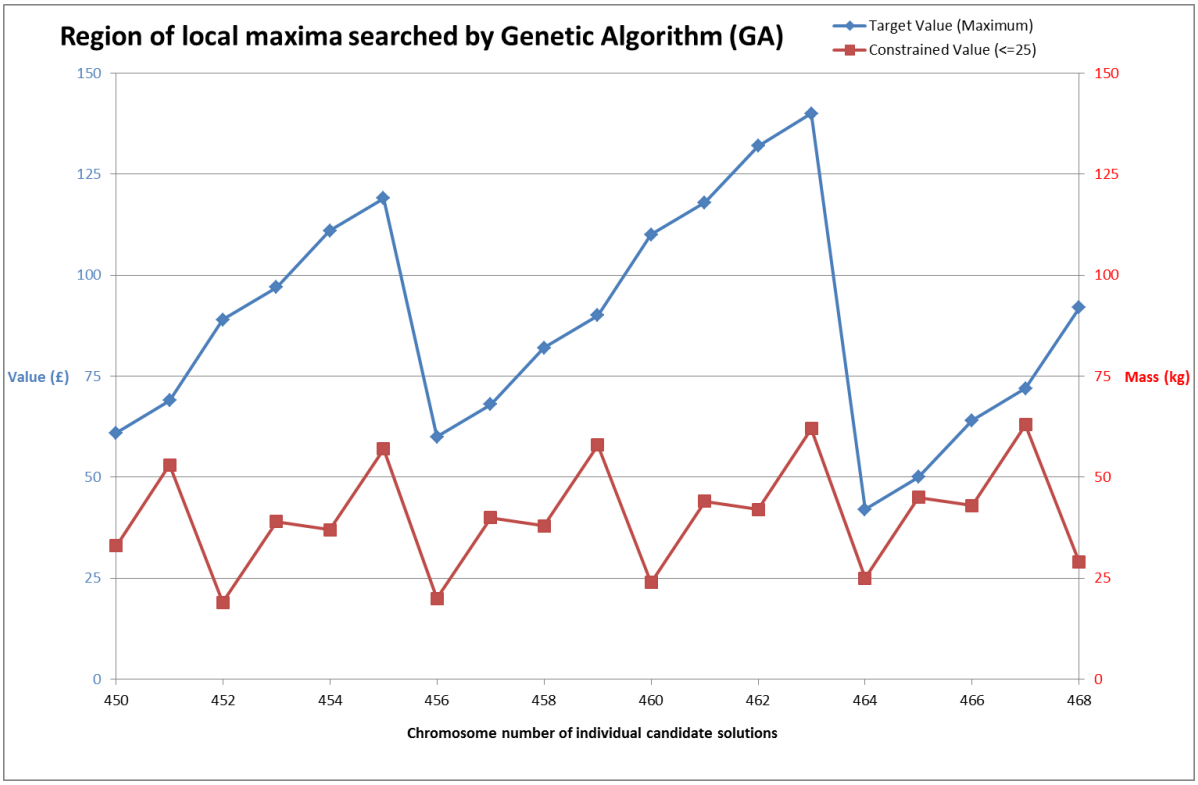


Figure 3-15 : The greatest achievable value for under 25kg is y=110 at x=460

3.5.2 Chromosomes for the ECU problem

In the following example, a binary valued matrix is used to map the allocation of a selection of five ECUs' resources to a similar number of features. This is to be scaled to manage any number. Features of the vehicle, for example lights, window-winding, immobiliser, alarm, are assigned to a specific ECU by number. There is a one-to-many relationship between the ECUs and the features such that each feature will have only one ECU but each ECU if selected may have one or more features. For example, an array representing the pairings of five features to ECUs could be

features_assigned to _ECUs[0 2 0 0 4] meaning that

feature[0] has been assigned to ECU[0],
feature[1] to ECU[2],
feature[2] to ECU[0],
feature[3] to ECU[0] and
feature[4] to ECU[4].

The representation of this problem as a single row vector means that the values of each element in the vector can be greater than 1 and this disallows some very useful binary arithmetic that could be used to help solve the problem. The same mapping of ECUs to features could be represented by the matrix,

$$C = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

so that the column number is the feature ID (zero to 4), the row number is the ECU ID (also zero to 4) and a non-zero entry marks the pairing of an ECU and a feature. Whilst there are now significantly more array elements that need to be stored in memory to represent the pairings, the algorithm for allocating a single ECU to support all features may begin with the single row chromosome, that is row zero = R_0 = [C_0=1, C_1=1, C_2=1, C_3=1, C_4=1, … ,C_n=1]

and continue with the systematic, deterministic, testing of chromosomes with a complete row of non-zero values, that is

row 1 = R_1 = [C_0=1, C_1=1, C_2=1, C_3=1, C_4=1, … ,C_n=1]
row 2 = R_2 = [C_0=1, C_1=1, C_2=1, C_3=1, C_4=1, … ,C_n=1]
row 3 = R_3 = [C_0=1, C_1=1, C_2=1, C_3=1, C_4=1, … ,C_n=1]
row 4 = R_4 = [C_0=1, C_1=1, C_2=1, C_3=1, C_4=1, … ,C_n=1]
. . . .
. . . .
. . . .
row n = R_n = [C_0=1, C_1=1, C_2=1, C_3=1, C_4=1, … ,C_n=1]

until the first row that does not support all of the features. If all of the subsequent ECUs in the data list have equal or less capacity compared with the previous ones, no more single-ECU solutions can be found. This allows the single ECU solution with the most appropriate ECU (measured by capacity/headroom or cost for example) with ECU data ordered so that the algorithm would stop as soon as the first unsuitable ECU (too small to accommodate all of the features) was found.

The algorithm could at this point move on to randomised crossover and mutation

This is just one of the possible default initialisations that might produce the optimal solution. Another would be to begin with a diagonal matrix representing a one to one mapping of a single ECU to each feature, thus,

D = [D_{00}=1, D_{11}=1, D_{22}=1, D_{33}=1, D_{44}=1, … , D_{nn}=1]

This starting point can be compared with the single row vector and the better chromosome of the two used as the first chromosome for the rest of the execution. These unique solutions are at the extremes of minimum and maximum use of ECUs for the number of features. Whilst the diagonal chromosome always produces a feasible solution (an ECU should be chosen to support each individual feature before the GA is executed) the single row vector is not guaranteed to have the capacity to support all features but it is a possibility and so it is tried first.

The GA is not suggesting methods of optimising the number of features and their requirements (speed, memory, etc.) so that an optimum number of ECUs can be prescribed.

Rather, it is taking the available resources and minimising the ECU usage based on the limitations of ECU and the constraints of the features' requirements.

Whilst the algorithm is coded so as to consider fewer ECUs to be a better fitness score, limitations and constraints could be placed on ECU usage to seek, for example, the best solution with no fewer than three ECUs.

3.5.3 Properties of each ECU and feature

For example, a vehicle braking system with the minimum number of required features might contain three components with a need for storage space required for the executable code (ROM size in kB), the necessary communication speed for transmission of data (bits per second for example) and a minimum processor speed (MHz) to execute the code in a timely fashion.

The data structure that represents the features and ECUs would be a pair of lists, one with rows of feature data and the other with rows of ECU data, looking like those in table 3-7 but without the column headings (not stored with the data) to represent the IDs and variables/attributes. The 'C' source code for this representation of the data is a 'struct' type and is set out in Appendix C and Appendix D.

Table 3-7 : Example data structure to represent a system of features and resources (ECUs)

ID	Name	ROM_size	com_speed	processor_speed	
0	"Controller "	4096	4096	4096	
1	"Calliper "	2048	2048	2048	
2	"Brake pedal"	1024	1024	1024	
ECU_ID	ECU_name	ECU_ROM_size	ECU_com_speed	ECU_proc_speed	ECU_cost
0	"ECU_0"	8912	8192	4096	1
1	"ECU_1"	8912	4096	4096	1
2	"ECU_2"	8912	4096	4096	1

There is a binary valued vector that describes the mapping from vehicle features (executable software code) to electronic control units (ECUs) by considering the elements of the vector as the elements of a square matrix. For example, in a system of five vehicle features, a 25 element vector, $v = [v_1, v_2, v_3, v_4, \dots, v_{25}]$, contains the binary values, $v_i \in \{0,1\}$ as representations of which ECU is paired with and supports which feature(s). Zero indicates no pairing whilst 1 represents an active pairing of ECU and feature. That is, the vector, v, in matrix or array form

$$v = \left(v_{1,}v_{2,}v_{3,}v_{4,}v_{5,}v_{6,}v_{7,}v_{8,}v_{9,}v_{10,}v_{11,}v_{12,}v_{13,}v_{14,}v_{15,}v_{16,}v_{17,}v_{18,}v_{19,}v_{20,}v_{21,}v_{22,}v_{23,}v_{24,}v_{25}\right)$$

$$V = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & v_5 \\ v_6 & v_7 & v_8 & v_9 & v_{10} \\ v_{11} & v_{12} & v_{13} & v_{14} & v_{15} \\ v_{16} & v_{17} & v_{18} & v_{19} & v_{20} \\ v_{21} & v_{22} & v_{23} & v_{24} & v_{25} \end{bmatrix} = A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,3} & a_{4,5} \\ a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} \end{bmatrix}$$

where $a_{ij} \in \{0, 1\}$ and i is the identifier of the ECU and j is the identifier of the feature thus

Table 3-8 : Example of one possible choice of mapping from features to ECUs for 5x5 problem

		Vehicle Features				
		1	2	3	4	5
ECUs	A	Paired	-	-	-	-
	B	-	Paired	Paired	-	-
	C	-	-	-	Paired	Paired
	D	-	-	-	-	-
	E	-	-	-	-	-

The vector, v, is therefore a binary string that can represent a chromosome to be manipulated by a genetic algorithm. Traditional GA operations such as crossover and mutation can be performed so as to generate offspring that may or may not satisfy constraints upon multiple non-zero values within either a column or row of the mapping matrix.

These constraints can be imposed during the random generation of chromosomes and at the crossover and mutation stages to prevent infeasible chromosomes from ever being generating or needing to be tested.

One question is whether eliminating the possibility of generating infeasible chromosomes is preferable to generating them, testing them and discarding them. Computation time is the only real factor in this question and the algorithm to prevent their generation needs to be weighed against the algorithm for sifting them.

If they are simply never generated (by ensuring or testing that the sum of the column entries is exactly equal to one) this precludes any need to execute a function that goes on to test whether a suitable candidate solution has been produced with respect to the capacity of the ECU and the required memory and performance attributes demanded by the feature that it supports.

This raises an issue with the types of mutations or crossovers and their effectiveness in converging on an optimal candidate solution. Many of the chromosomes that could be generated by random mutations or crossovers would simply be invalid with respect to a physical mapping of ECU to feature in such a predictable way that generating the chromosome only to be discarded by the fitness function would be a waste of processor time.

For example, any chromosome in which there are multiple non-zero values in any column represents an impossible mapping.

The features are distinct and cannot be shared across more than one ECU, so more than one non-zero value in a column would be attempting to represent a single feature that is on two separate ECUs. Although the code for a single feature could be modularised so as to run either in parallel as two distinct pieces of code or as two distinct executable programs, this would then become two separate features with respect to the representation in the GA vector.

That is table 3-9 would occur by the generation of the chromosome,

V = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1]

Table 3-9 : Example of an implementation of features to ECUs for 5x5 problem

		Vehicle Features				
		1	2	3	4	5
ECUs	A	1	0	0	0	0
	B	1	0	0	0	0
	C	0	1	0	0	0
	D	0	0	1	0	0
	E	0	0	0	1	1

and if the this mapping were actually possible by decomposing vehicle feature number 1 into two separate modules of executable code, feature number 1 would instantly need to become two distinct entities with their own unique IDs and the resulting representation would be the one in table 3-10

Table 3-10 : Example of an implementation of features to ECUs for 6x6 problem

		Vehicle Features					
		1	2	3	4	5	6
ECUs	A	1	0	0	0	0	0
	B	0	1	0	0	0	0
	C	0	0	1	0	0	0
	D	0	0	0	1	0	0
	E	0	0	0	0	1	1

This would be planned for, before execution of the GA and six ECUs would be available for the six features represented in a 6x6 matrix or a 36 element vector so that the only valid chromosomes would be those with exactly one non-zero value in every column and every feature could have its own ECU if necessary. Any column with only zero entries would be invalid because this would mean that a particular feature was not supported by any ECU.

In the specific cases of using as many ECUs as features that need to be supported, the GA will examine a square matrix of possible chromosomes and will begin with the identity matrix representing a single ECU for each individual feature, thus…

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

so that, for three features supported on three separate ECUs, the representation shows ECU[0] supporting feature[0], ECU[1] supporting feature[1] and ECU[2] supporting feature[2].

Once this candidate solution has been investigated and found to be acceptable (there shouldn't be a scenario in which this is not at least a feasible solution) the GA will examine the possibility of supporting all of the features on just one ECU.

$$R_1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad R_2 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \qquad R_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

The three representations show either the first ECU supporting every feature, the second ECU supporting every feature or the third ECU supporting every feature. The GA is likely to reject latter chromosomes where the smallest or least powerful ECU tries to support all of the features.

Conversely, if the largest and most powerful ECU cannot support all of the features, the GA does not need to examine the other row vectors that represent features supported by a single ECU, as these should be ordered by size/power in such a way that once the first row is reached that is not a feasible solution, no row below it in the matrix should be able to represent a feasible solution.

Once these chromosomes have been examined, the GA proper will execute with the less deterministic, random generation of subsequent offspring. The fitness score is based on minimising the number of ECUs used, so that a solution requiring only one ECU will trump all other solutions. Other criteria could be programmed into the algorithm to limit the minimum number of ECUs used if there were a preference for not solving the problem on a single ECU. The lower limit could be set as any value and a solution that used fewer than this number of ECUs would be rejected by the algorithm as unfit.

A solution that can accommodate all of the features, but with large unused overheads of resources on the ECU(s), is not currently considered to be a better solution than one that fully utilises the capacity of all of the ECUs. This could be programmed into future versions of the software if desired if, for example, using a smaller ECU which can still accommodate the features has some cost saving benefit.

To avoid ever generating invalid chromosomes (ones that produce a conflict in the system they are attempting to represent) all stages of generation, crossover and mutation need to have rules built in that apply constraints on the row and column values of the vector and matrix. This means that any instances of mutation or crossover, despite having relatively small probabilities of ever occurring, would produce quite different offspring from the parents, for example, for a parent pair in a system of just three features, there are only 27 valid, feasible, candidate solutions, represented by the chromosomes. They are not evenly

spaced or in any recognisable sequence that can be generated by a function. Each successive value, when evaluated as the base ten equivalent of the binary representation, is either seven greater than the previous value or greater by multiples of seven up to 56 greater than the previous value in the sequence. The sequence of binary chromosomes

[000000111], [000001110], [000010101], [000011100], [000100011],

[000101010], [000110001], [000111000], [001000110], [001010100],

[001100010], [001110000], [010000101], [010001100], [010100001],

[010101000], [011000100], [011100000], [100000011], [100001010],

[100010001], [100011000], [101000010], [101010000], [110000001],

[110001000], [111000000]

has the denary values

7, 14, 21, 28, 35, 42, 49, 56, 70, 84, 98, 112, 133, 140, 161, 168, 196, 224, 259, 266, 273, 280, 322, 336, 385, 392, 448

and the sequence of differences between each successive term is

7, 7, 7, 7, 7, 7, 7, 14, 14, 14, 14, 21, 7, 21, 7, 28, 28, 35, 7, 7, 7, 42, 14, 49, 7, 56

making it intractable to formulate a generating function from the n^{th} term of the sequence.

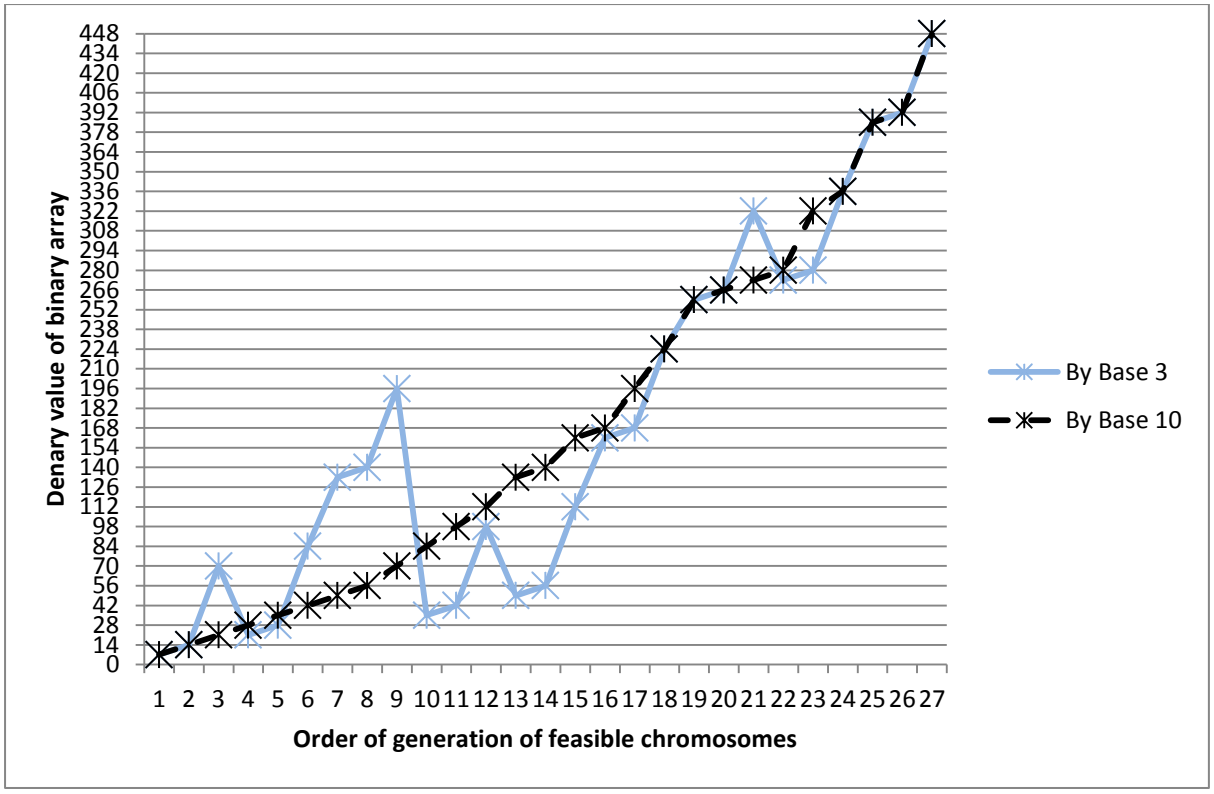


Figure 3-16 : Graph of the generation of binary sequence in base 3 and base 10

For any of the chromosomes to mutate into another feasible chromosome that would not be instantly rejected, simply 'switching off' one gene or 'switching on' another will not necessarily result in a feasible solution. Unlike the knapsack problem where every possible chromosome represents some packing, whether too heavy or not valuable enough, the ECU chromosome can produce mappings that are simply not possible. For example, many ECUs might be allocated the same feature or some features might not be supported at all. Whilst these can be tested, it is a waste of time producing them if a preliminary test can determine their usefulness or otherwise by a simple check of the column sums of the chromosome as a matrix.

Whilst it is obvious to see that a change in just one element of the vector, from zero to one or vice versa, is a relatively small difference in the overall pattern of the elements, there is no apparent method to anticipate the difference in the effect on the value of the fitness score or the output of the fitness function that this could have. A chromosome which is close in appearance to one that produces a viable solution may produce its own solution that is vastly far away from the ideal solution. Therefore, it is not the score of the chromosome itself that can be measured but the score of the solution that it produces. To infer that a chromosome, because it is like another one, has any merit with respect to finding a solution is normally erroneous.

The problem with the discretised GA is that a candidate solution may be very close to the currently chosen best solution but may bear so little resemblance to it in terms of the chromosome by which it is represented that the algorithm itself isn't directed towards the best one, by the current chromosome or by the current best solution.

Take, for example, a dart board or a roulette wheel. If the aim is to score 180 with three darts, each scoring 'treble 20s', how close to that aim are three 'treble 1s'?

In terms of the score, the difference is about as big as it can be at 171, but the proximity of the darts to a perfect score could not be closer without achieving the aim.

Similarly, moving around the board just one more sector, three 'treble 18s' would record a score of 162 and is therefore both close in terms of position on the board as well as points scored, with respect to scoring 180. Because the position of each score around the board is known (the 20 sector is always at the twelve o'clock position sandwiched between five on the left and one on the right) it is relatively easy to determine how far away any given score is away from the maximum, in terms of both numeric value and physical proximity.

But what would any algorithm be able to produce, if the positions of the numbers were only revealed after a dart had struck the board? How would a score of three inform any algorithm of where the 20 was on the board?

The approach can be improved by knowing all of the possible solutions beforehand and/or ordering them from lowest to highest (that is converting the dart board so that the scores for each sector in between one and twenty going clockwise are in numerical order). In the case of throwing a single dart, a 'treble 1' would immediately indicate the need to aim again at the sector to the left, even if the board were rotated and the absolute positions of each score were not known, there would be clues as to the relative positions as soon as the first dart is thrown and the score revealed.

This is not possible if the relative positions of each score are random or, in some arbitrary way, non-consecutive. The current GA is successful in finding solutions where they exist but it is very sensitive to the starting conditions. For example, for the 5 x 5 matrix, initialising the chromosome with the identity matrix

$$I=\begin{bmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{bmatrix}$$

will represent a worst feasible solution based on a 1:1 mapping of specifically ordered ECUs and features, manually selected to provide sufficient resources for the requirements demanded by the features. On occasions this will also be the best solution.

When the chromosomes are initialised with all of the non-zero values in the first row, known to be an invalid solution to the problem based on memory capacity, for example the first ECU is not capable of supporting the full set of features but another ECU in a different row is, the GA does evolve the chromosome to arrive at a solution where a single row is populated with non-zero values. Ideally, the first ECU (ECU 0) should be the one with the greatest capacity but this is not always possible to achieve for systems with more than one variable.

Specifically, for ECUs with memory capacities of {1000, 100000, 100000, 2048, 4096} and features with memory requirements of {512, 512, 512, 512, 512} it's clear that the first ECU cannot support the total of 2560 (units not specified) but any of three other ECUs can.

In this case, the GA returns the matrix

$$R_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

representing a mapping of the second ECU (memory capacity 100,000) on to all features.

The two extremes of ECU usage are either a single ECU supporting all features or one ECU per distinct feature. Using these as initialisations for the parent chromosomes means that any improvements found during execution will produce a less extreme solution. The first comparison will be to test whether either of the extremes or one of the offspring of the extreme parents can support all features. If none of them supports all of the features, the next generation parents will be from the two chromosomes that support more features than the others. Execution will stop when either no more improvements can be expected (if all of the children and the parents converge to the same chromosome after an arbitrary number of iterations).

The purpose of the genetic algorithm is to constrain the number of candidate solutions to a manageable size so that the execution will terminate before such a period of time that would make the project too costly in terms of time to market.

3.5.4 The use of number base systems to reveal the shadows of higher dimensions

In the system of ECUs and features where there is a possibility of supporting each feature with its own individual ECU, a square matrix of values represents the mapping of each ECU to a feature with the identity matrix, thus, in binary

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

so that, for three features supported on three separate ECUs, the representation shows ECU[0] supporting feature[0], ECU[1] supporting feature[1] and ECU[2] supporting feature[2].

For this system (and for all other equal numbers of ECUs/Features) the sample space of all possible matrices with binary valued elements is equal to all of the unique ways in which 0s and 1s can fill the *n* x *n* matrix in unique ways. This is the same as treating all of the separate rows of the matrix as a single 1 x nm vector that can represent any binary value from zero to $[n_1 = 1, n_2 = 1, n_3 = 1 \ldots n_m = 1]$, for example [111111111] for the 3x3 matrix representing three features and three ECUs.

Since there are n^2 elements in the square *n* x *m* matrix and since the total number of binary representations if all of these elements fill a $1\ x\ n^2$, binary valued vector, it is clear by inspection that there are $2^{(n^2)}$discrete values that the matrix can adopt, including the zero matrix, where *n* is the number of features in the system. This *n* is also equal to the maximum number of ECUs, the number of rows in the matrix and the number of columns in the matrix. However, many of the possible binary representations do not represent solutions that should even be considered. Discounting those matrices where there is greater or fewer than one non-zero value in any column(s) the total number of feasible representations that should be tested is

$$\prod_{i=1}^{n} m_i = m^n = n^n, where\ n = m \qquad (3\text{-}4)$$

where i is the column number, m_i is the number of rows in i and n is the number of columns. For example, the following matrices should all be absent from the set of possible matrix representations.

$$X=\begin{bmatrix}1 & 1 & 1\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix} \qquad Y=\begin{bmatrix}1 & 0 & 0\\ 1 & 1 & 1\\ 0 & 0 & 0\end{bmatrix} \qquad Z=\begin{bmatrix}1 & 0 & 0\\ 1 & 0 & 0\\ 1 & 1 & 1\end{bmatrix}$$

But to remove these matrices after they have been generated means creating disproportionately more infeasible matrices than feasible ones, adding to program execution time which increases exponentially as the system grows in size. It is not immediately obvious whether there is a generating function that can produce only those matrices with exactly one non-zero element in each column but there is a numerical method, provided by this research and implemented in software that does perform this task.

Two useful cases to examine are those of a 2x2 and a 3x3 chromosome. Examining the case of two features, both can be shared, in two different ways, either on one ECU or the other. They may be supported by a single ECU each and, if the capacity of the ECUs allows it, the pairing may be swapped so that each is supported by the other ECU instead. The feasible matrices from a total of 16 are

$$A=\begin{bmatrix}1 & 1\\ 0 & 0\end{bmatrix} \qquad B=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \qquad C=\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \qquad D=\begin{bmatrix}0 & 0\\ 1 & 1\end{bmatrix}$$

and considering the transpose of each column as a binary vector in its own right, the sums of the vectors' elements have unique values for any representation of 1s and 0s. For example, if the first column of A is transposed to become the vector [1 0] and if the elements are read as though they are a single binary number, that is 10_2 then the value of the first column of A is 2_{10} as is the sum of the second column of A, the first column of B and the second column of C. Similarly, the second column of B, the first column of C and each column of D is each equal to 1_{10}, being the binary representation 01.

So, the sequence from A to D is [1 0],[1 0]; [1 0],[0 1]; [0 1],[1 0]; [0 1],[0 1] treating each new vector as the transpose of each column giving the base 10 values, 2,2 ; 2,1 ; 1,2 ; 1,1 and by symmetry the matrices could be labelled D,C,B,A to give the sequence 1,1 ; 1,2 ; 2,1 ; 2,2

This, latter, sequence is useful as the basis for a sequence generating algorithm for any size of matrix. Consider the 3x3 case, in which there are 512 possible matrices of 1s and 0s but only 27 that satisfy the criteria for feasibility. By eliminating 485 infeasible matrices before starting, a huge reduction of computer processing time is achieved.

The 27 feasible matrices are

$$\begin{bmatrix}1&1&1\\0&0&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&1&1\\1&0&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&1&1\\0&0&0\\1&0&0\end{bmatrix}\begin{bmatrix}1&0&1\\0&1&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&0&1\\1&1&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}\begin{bmatrix}1&0&1\\0&0&0\\0&1&0\end{bmatrix}\begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}\begin{bmatrix}0&0&1\\0&0&0\\1&1&0\end{bmatrix}$$

$$\begin{bmatrix}1&1&0\\0&0&1\\0&0&0\end{bmatrix}\begin{bmatrix}0&1&0\\1&0&1\\0&0&0\end{bmatrix}\begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix}\begin{bmatrix}1&0&0\\0&1&1\\0&0&0\end{bmatrix}\begin{bmatrix}0&0&0\\1&1&1\\0&0&0\end{bmatrix}\begin{bmatrix}0&0&0\\0&1&1\\1&0&0\end{bmatrix}\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}\begin{bmatrix}0&0&0\\1&0&1\\0&1&0\end{bmatrix}\begin{bmatrix}0&0&0\\0&0&1\\1&1&0\end{bmatrix}$$

$$\begin{bmatrix}1&1&0\\0&0&0\\0&0&1\end{bmatrix}\begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}\begin{bmatrix}0&1&0\\0&0&0\\1&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}0&0&0\\1&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}0&0&0\\0&1&0\\1&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\0&0&0\\0&1&1\end{bmatrix}\begin{bmatrix}0&0&0\\1&0&0\\0&1&1\end{bmatrix}\begin{bmatrix}0&0&0\\0&0&0\\1&1&1\end{bmatrix}$$

Considering each row of any matrix as a 3-digit binary value, the maximum value that any row can take is 7 and if such values for each of the three rows in any matrix are considered as a co-ordinate triple, the values can be graphed to give values in an xyz axis system. Values for the first nine matrices above would be…

(7,0,0), (3,4,0), (3,0,4), (5,2,0), (1,6,0), (1,2,4), (5,0,2), (1,4,2), (1,0,6)

and to plot all 27 matrices as single points in 3-space, we would assign the three vectors; x y & z thus

x=[0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 3 3 4 4 4 4 5 5 6 6 7];

y=[0 1 2 3 4 5 6 7 0 2 4 6 0 1 4 5 0 4 0 1 2 3 0 2 1 0 0];

z=[7 6 5 4 3 2 1 0 6 4 2 0 5 4 1 0 4 0 3 2 1 0 2 0 0 1 0];

which clearly shows the alignment of the xyz values (7,0,0), (3,4,0) etc., as above.

The distribution of the feasible chromosomes makes a symmetrical and self-similar arrangement of triangles when plotted in 3 dimensions. The points can be plotted as in figure 3-17 and all lie in a plane that is a larger triangle bounded by the lines described by the parametric equations

$$\begin{cases} x = 7 - y \\ z = 0 \end{cases} \tag{3-5}$$

$$\begin{cases} y = 7 - z \\ x = 0 \end{cases} \tag{3-6}$$

$$\begin{cases} z = 7 - x \\ y = 0 \end{cases} \tag{3-7}$$

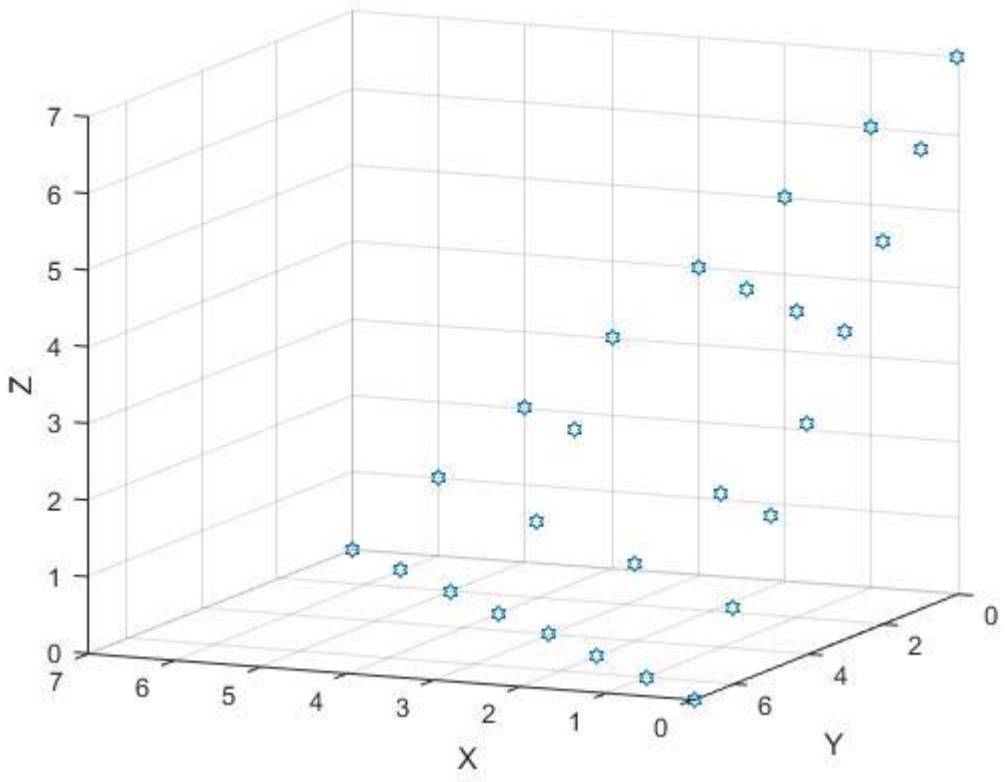


Figure 3-17 : Cartesian coordinates of the 3-D matrices of the 3 feature problem

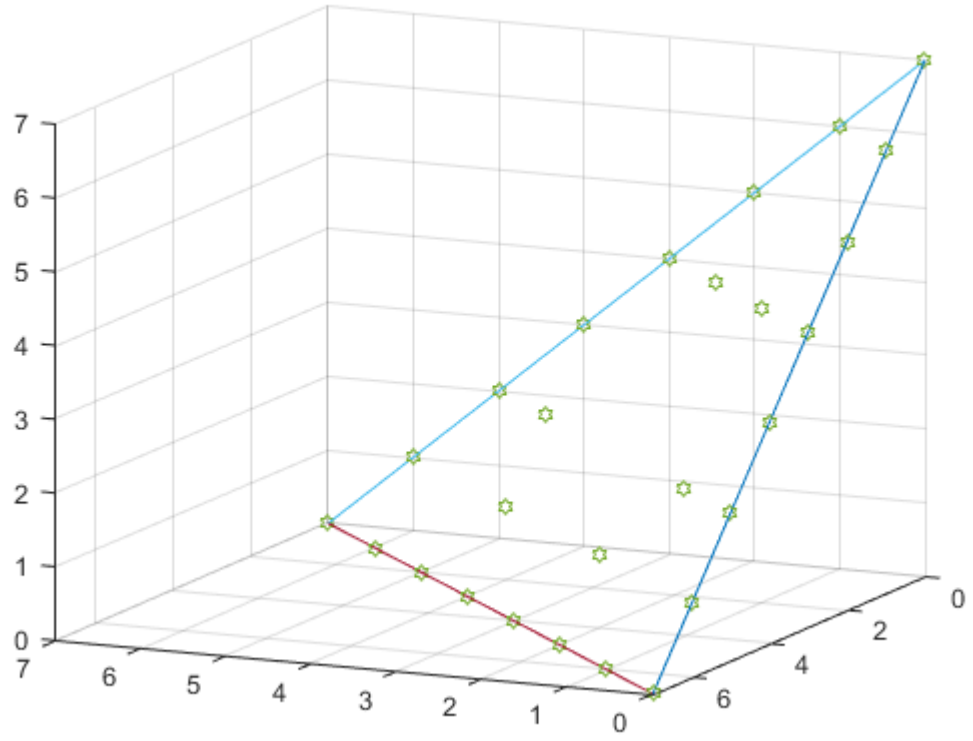


Figure 3-18 : A plane figure in 3-Space representing a two dimensional 3-D geometry

$$\begin{cases} x = 0 \\ y + z = 7 \end{cases} \quad (3\text{-}8)$$

$$\begin{cases} y = 0 \\ x + z = 7 \end{cases} \quad (3\text{-}9)$$

$$\begin{cases} z = 0 \\ x + y = 7 \end{cases} \quad (3\text{-}10)$$

For the 4-D model, it is shown by this research that all of the points exist to form a 3-D tetrahedron, one face of which is plotted in 3-D in table 3-21 and as a 2-D projection in table 3-22.

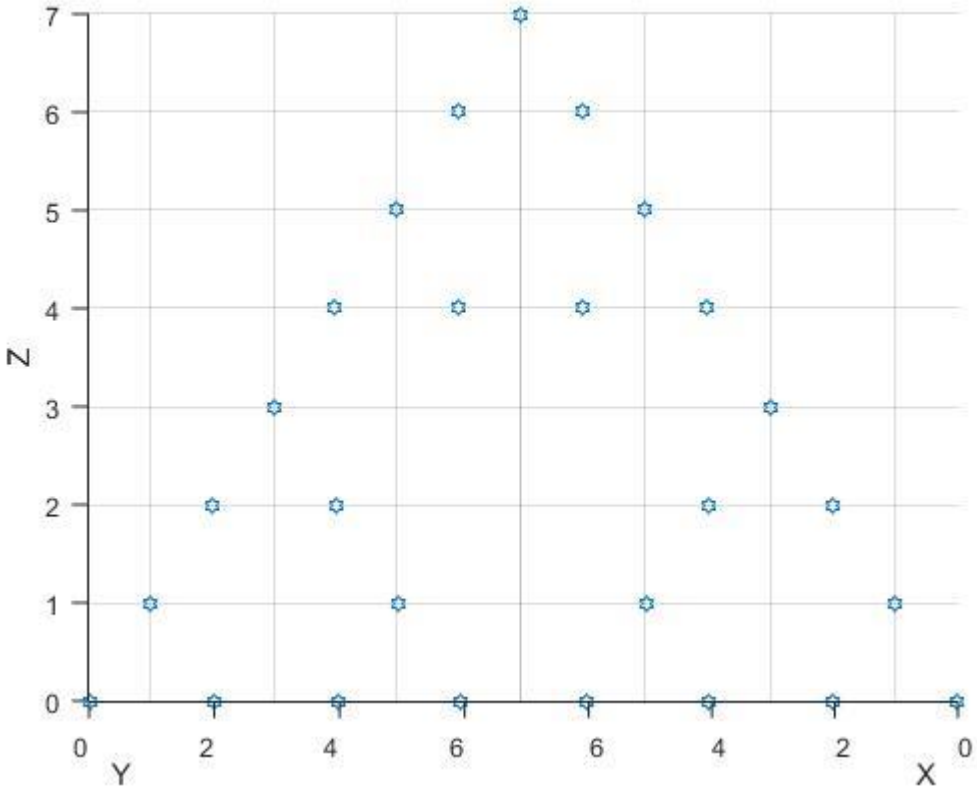


Figure 3-19 : Coordinate points of a plane figure representing the feasible chromosomes of a 3x3 ECU/features problem

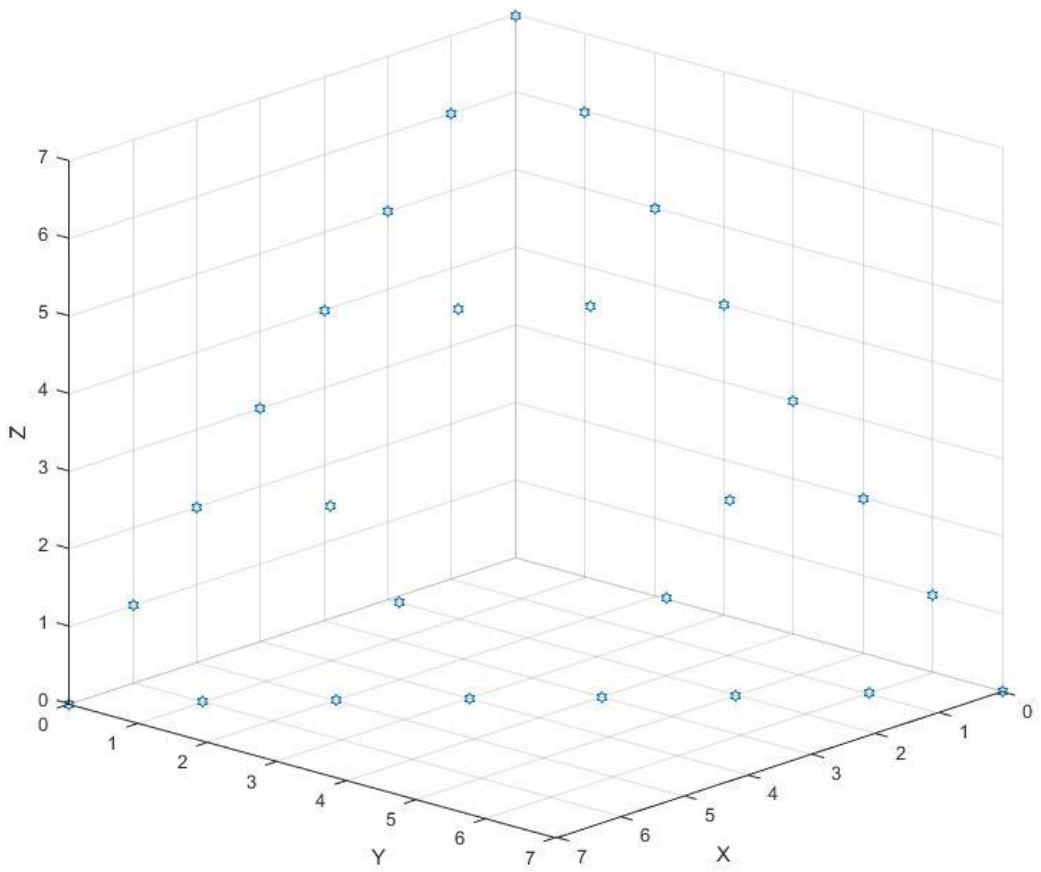


Figure 3-20 : Self similar triangular planes of feasible space for 3x3 chromosomes

But this is still only possible to obtain by filtering from all the points (0,0,0) to (7,7,7) which is 8^3 points or 2^9 = 512.

To generate only the values that create the feasible matrices, one needs to think in the number base that is equal to the number of rows or columns in the matrix. That is, if the matrix is 3x3, one must consider the base 3 number system.

Starting with a 3x3 matrix with a completely non-zero first row and zeros elsewhere, a vector is constructed that records only the row numbers of the non-zero elements for each column.

For example, for the matrix $\begin{bmatrix}1 & 1 & 1\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$ a vector $[0 \quad 0 \quad 0]$ records that the non-zero element

in column zero resides in row zero, the same as for columns 1 and 2.

For the 3x3 diagonal matrix, a vector can record that the first, second and third rows of each respective column are non-zero, that is for the matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The vector $[0 \quad 1 \quad 2]$ records that column zero has a non-zero entry in row zero, column 1 has a non-zero entry in row 1 and column 2 has a non-zero in row 2. Following this, all of the feasible values for the ECU/feature matrix can be represented by different values of a 1x*n* vector, for example

$[0 \quad 0 \quad 0][0 \quad 0 \quad 1][0 \quad 0 \quad 2][0 \quad 1 \quad 0][0 \quad 1 \quad 1][0 \quad 1 \quad 2][0 \quad 2 \quad 0][0 \quad 2 \quad 1][0 \quad 2 \quad 2]$

$[1 \quad 0 \quad 0][1 \quad 0 \quad 1][1 \quad 0 \quad 2][1 \quad 1 \quad 0][1 \quad 1 \quad 1][1 \quad 1 \quad 2][1 \quad 2 \quad 0][1 \quad 2 \quad 1][1 \quad 2 \quad 2]$

$[2 \quad 0 \quad 0][2 \quad 0 \quad 1][2 \quad 0 \quad 2][2 \quad 1 \quad 0][2 \quad 1 \quad 1][2 \quad 1 \quad 2][2 \quad 2 \quad 0][2 \quad 2 \quad 1][2 \quad 2 \quad 2]$

…which are simply the ternary values of the base ten numbers from zero to 26 (the twenty seven values that correspond to the feasible matrices with exactly one non-zero entry in every column.

Without needing to draw the 4-D model, a vector of four elements (in base 4) beginning with [0 0 0 0] and ending with [3 3 3 3] can represent the $3x4^3 + 3x4^2 + 3x4^1 + 3x4^0$ different matrices in a 4x4 system + the zero vector = 256 different matrices which plots a 3-D shadow of the 4-D space. And so on, the number of elements (n) in a vector of base n will generate exactly the $n^{(n)}$ matrices that record the non-zero rows of each column of the ECU/features matrix.

For the 4-D model that attempts to solve a 4x4 matrix of ECU/feature pairs, a representation in 3-D exists that is bound by the intersection of four planes in 3-D forming a regular tetrahedron with sides of length 15. Taking the 4-D coordinates (w, x, y, z) that have a zero value in the ‘w’ axis, one face of the tetrahedron can be plotted, to include 81 of the 256 points. Holding each of the other three axes’ values constant and equal to zero in turn, produces a graph of each of the other three triangles. The entire 4-D solid has a shadow in 3-D that can be built in 3-D or represented on an x, y, z coordinate system and plotted in 2-D.

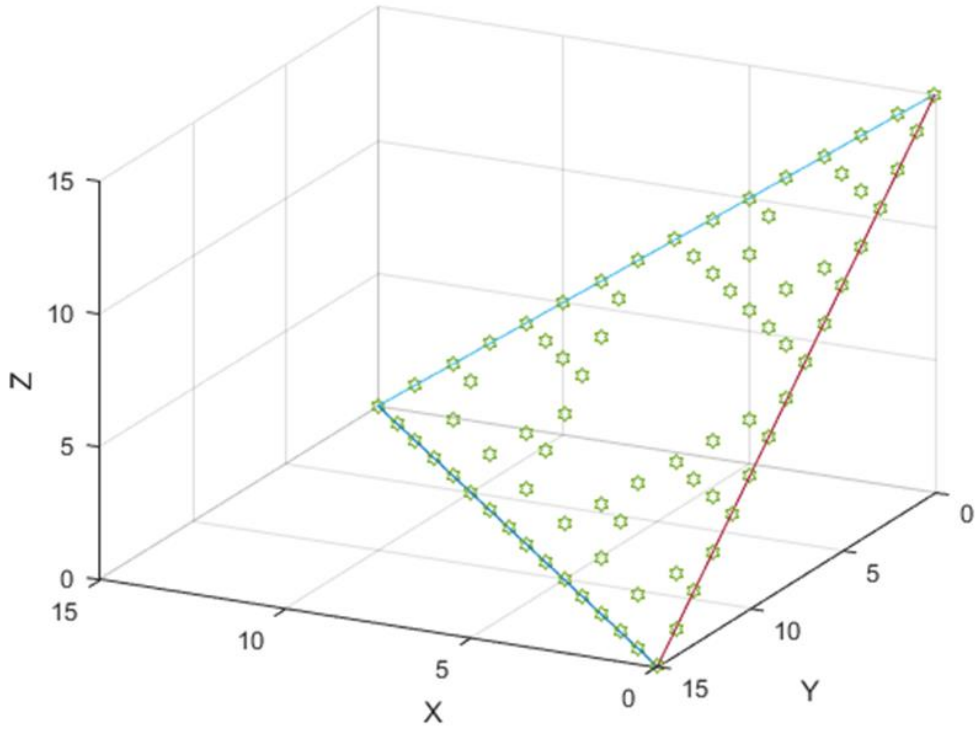


Figure 3-21 : 81 points that describe one face of a tetrahedron in the 4x4 chromosome

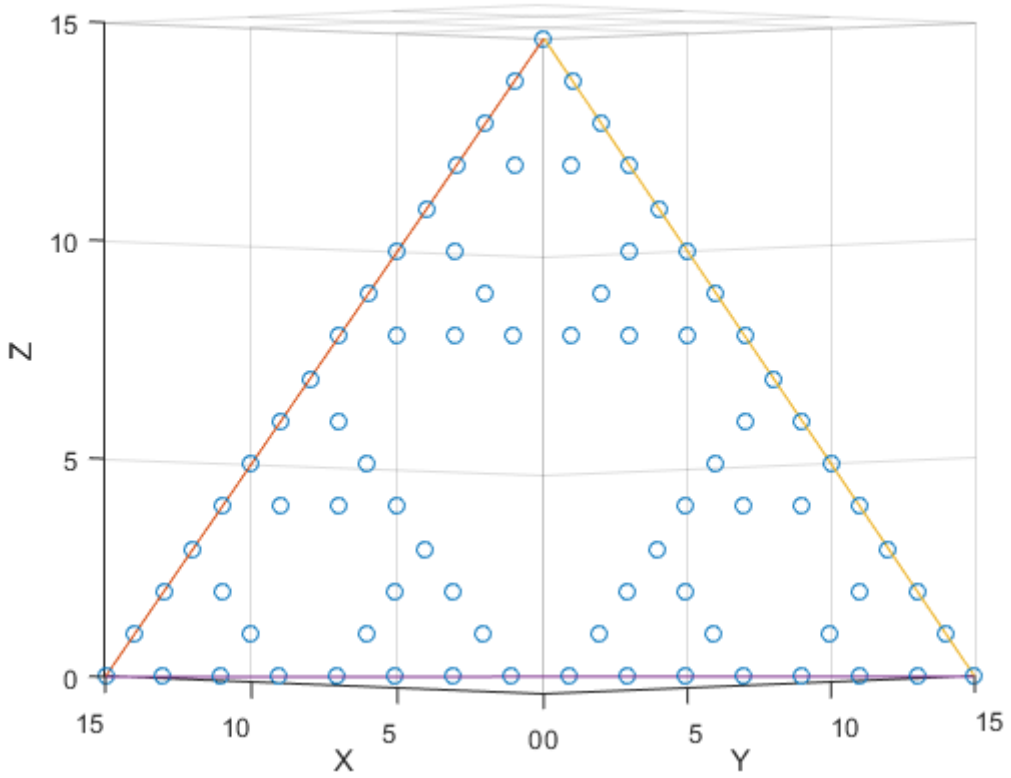


Figure 3-22 : Rotated 2-D plan view of the self-similar triangles and coordinate points representing a face of the tetrahedron that exists in 4 dimensions

$$\begin{cases} w = 0 \\ x = 0 \\ y + z = 15 \end{cases} \quad (3\text{-}11)$$

$$\begin{cases} w = 0 \\ y = 0 \\ x + z = 15 \end{cases} \quad (3\text{-}12)$$

$$\begin{cases} w = 0 \\ z = 0 \\ x + y = 15 \end{cases} \quad (3\text{-}13)$$

In generating candidate solutions, the GA has so far only been programmed to find solutions with the fewest number of ECUs required while satisfying the criteria of supporting every feature of the system.

This means that a solution using only three ECUs, for example, will not be bettered by another subsequent solution that also uses only three ECUs, even if some other legitimate criterion such as ‘amount of spare capacity on ECUs’ or ‘cost to purchase’ would make the candidate solution better than the suggested one.

These specifics could be programmed into the GA but while they are currently not considered, it is possible that a solution offered by the GA is not the same one offered by the exhaustive search programme (ESP) whilst still being completely feasible and legitimate. One reason for this is the order in which the ESP works through the solution space in a methodical and deterministic, albeit arbitrary, way.

In one run, the random GA might find a specific one of the solutions that uses the least number of ECUs before it tests another one and the one that it finds first will not be overwritten as the algorithm only overwrites if a solution is better than all previously found solutions in that run.

However, on a subsequent run of the GA, the random nature of the generation of new offspring could produce a solution that is different but perfectly plausible in terms of the number of ECUs that are required, simply because of the order in which it generated and tested the offspring’s’ chromosomes.

This means that there may be more than one optimal solution but that the ESP will only offer one of them. The introduction of other criteria, such as the aforementioned ‘cost to purchase’ or ‘spare capacity’ would further constrain the possible solutions.

For example, in a 3x3 chromosome, the following may both be feasible and optimal according to the criteria programmed into the executable GA,

$$A=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 1\\ 0 & 0 & 0\end{bmatrix} \qquad B=\begin{bmatrix}0 & 0 & 0\\ 1 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

and the exhaustive model will first encounter A, since the third column (as in B) is the last to populate its third row. However, the random nature of the GA means that even if both the above chromosomes are generated, the order in which they are encountered in any two runs maybe different. The first one to be generated and tested will take priority over the second one because it wold only be replaced if the second one is considered to be an improvement, that is using fewer ECUs.

Manual examples can quickly be created to demonstrate scenarios where any mapping is optimal for the 2x2 system. Similarly, for a 3x3 system, with 27 different feasible ways of mapping the ECUs to the features, examples can be created for which the optimal mapping can be known in advance for validation of the GA.

For example, consider three features that require 6kB, 2kB and 1kB of memory on the ECUs supporting them. If there were, for example, three ECUs available that had memory capacities of 9kB, 8kB and 4kB respectively, the optimal arrangement of features with respect to memory alone would be to place all of them on the single ECU capable of supporting the total of 9k required by all of the features.

For ECUs and features that all have exactly the same memory requirement and capacity, that is three features requiring 4k each and three ECUs providing 4k capacity each, we would require one ECU for each feature and the matrix mapping would be any one of the following

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} \quad \begin{bmatrix}1 & 0 & 0\\ 0 & 0 & 1\\ 0 & 1 & 0\end{bmatrix} \quad \begin{bmatrix}0 & 1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 1\end{bmatrix} \quad \begin{bmatrix}0 & 1 & 0\\ 0 & 0 & 1\\ 1 & 0 & 0\end{bmatrix} \quad \begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{bmatrix} \quad \begin{bmatrix}0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}$$

Less trivial solutions for the remaining 21 feasible solutions include those where all of the features can be supported on any single ECU, for example where any single ECU has sufficient capacity to support all features, represented by any of the following

$$\begin{bmatrix}1 & 1 & 1\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} \quad \begin{bmatrix}0 & 0 & 0\\ 1 & 1 & 1\\ 0 & 0 & 0\end{bmatrix} \quad \begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 1 & 1 & 1\end{bmatrix}$$

The maximum possible count for the specific implementation of this program was 4,294,967,295 and so to be able to iterate this many times and more, nested loops equal to the cubed root of the maximum required count were set up.

That is for the 6x6 case, the maximum number of passes through the loop is 2^{36} which is possible with three nested loops of 4096 each. Because the loops only have to perform and do not need to keep a tally, 4096^3 iterations will be performed.

Because this number of nested loops cannot be determined at runtime – and to avoid having to hard code a different number of loops for every compilation of the code – a fixed number of loops is coded and the value for the size of each loop is assigned before compilation

The exhaustive algorithm calculates the fitness score of every possible combination of 0 and 1 in every element of the *n x n* GA array. This means that many infeasible chromosomes are being checked after they have been generated. For example, starting with the bottom row of the GA array, the algorithm counts to seven in its binary representation after first generating six unusable chromosomes in which at least one feature is not considered.

3.5.5 Scaling the GA problem

After considering the ESP/GA with limited number of iterations before termination (fewer than half of the ESP) an alternative method for determining convergence or termination of the program was developed by this research. It counts the number of iterations taken to produce a new solution since the previous one held in memory and each new solution informs the algorithm how long it should allow before the next new solution and terminates if this is exceeded. A graph of the number of iterations allowed and used to find each solution is shown in figure 3-23 for a single run of the GA with 99 features and maximum iterations of $2^{\wedge 31}$ = 2147483648

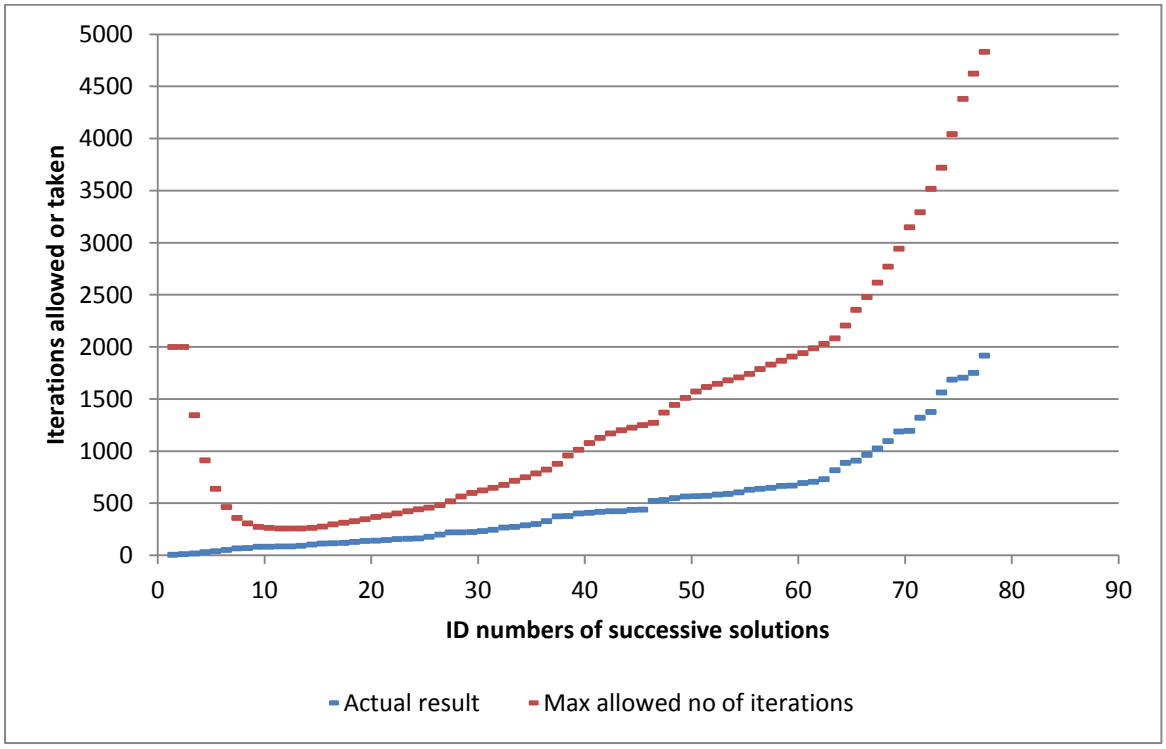


Figure 3-23 : Graph of variable limit to number of iterations allowed before next solution

Figure 3-24 shows successive improvements to the solution over time (number of iterations taken) for seven different executions of the GA. Each individual execution finds between 75 and 85 different and better solutions over the course of between 2000 and approximately 3500 iterations. The worst case, taking 3583 iterations to find the best solution before terminating after a total of 7983 iterations without a better solution found, executed in a time of 4.4 seconds to complete the algorithm. The 7983 iterations cause the program to terminate due to passing beyond a limit on the allowable number of iterations since the last known better solution. The graph demonstrates that there is no benefit in running the GA for longer in the hope that it will find one more improved solution as the trade off in time is too great.

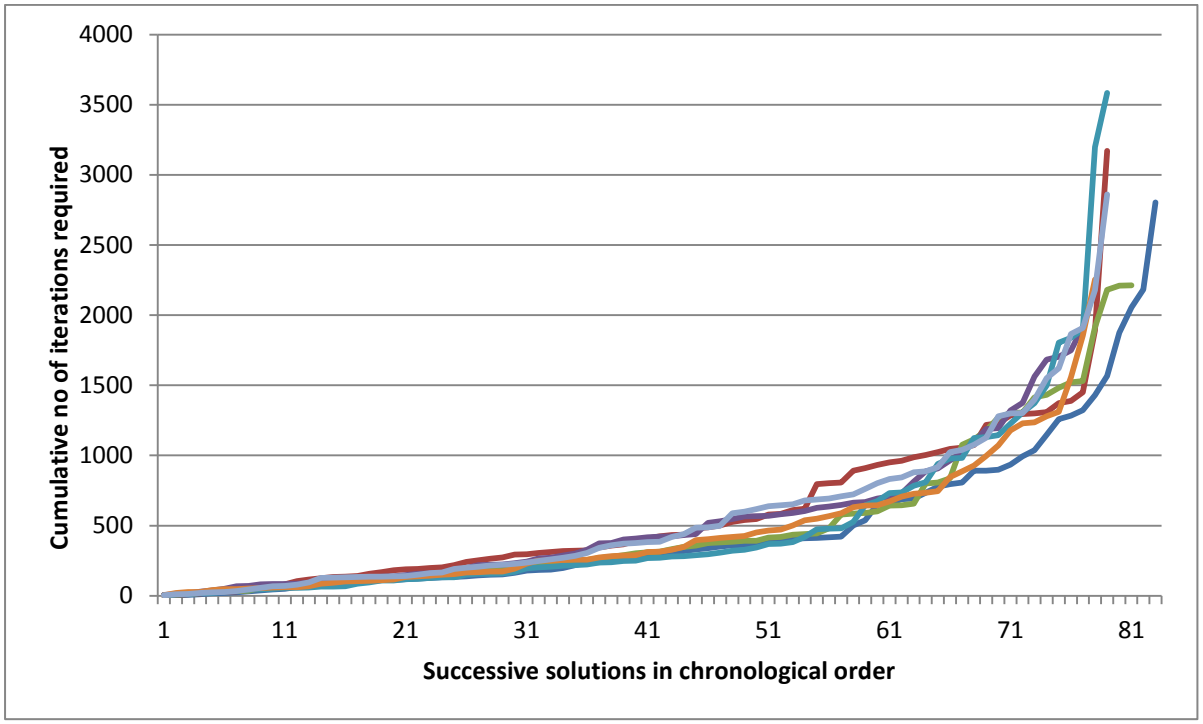


Figure 3-24 : Graph of cumulative iteration count for successive new chromosome solutions

The average time for a single run is 3.47 seconds and the average maximum number of iterations taken is 6266. The last iteration on which a new useful and better chromosome was found is 2686, on average.

3.5.6 GA with relatively low number of iterations before termination

In this model, the GA is programmed with an arbitrary maximum number of iterations unrelated to the size of the problem and based more closely on the amount of time that the program would take to execute considering the amount of computer processing time per iteration. This can produce useful sub-optimal results for very large numbers of features and ECUs by randomly searching a relatively small subset of the entire search space. However for very large sample spaces, it is more likely that the GA will not find the optimal solution. Ideally, a GA which will execute no more than half of the expected iterations of the ESP for the same problem is worth executing but this number is impractical for very large problems and the execution time becomes a function of the time available in which to find a solution.

3.5.7 Random neighbours of discrete data

The GA that solves the ECU problem cannot use information about its current location in the search space or the relative direction and gradient of regions nearby to move closer towards an optimal solution. Instead, the GA must conduct searches of subsets of the solution space determined by the results of changes to the chromosomes as a result of splicing sections of a pair of parent chromosomes or by random mutations, much like in a biological reproductive system that produces offspring with random variations that are tested for fitness in the child's lifetime. If the gradient (derivatives of the objective function) can be calculated or approximated, a steepest descent (or ascent) indicating the direction to take can be applied to move from one neighbour to another. Otherwise, a random or deterministic generation of a subset of neighbours is carried out [58] and, with a single mutation of just one gene in a chromosome, switching a zero valued bit on or turning a non-zero valued bit off, the genetic algorithm defines a new candidate solution that is a neighbour of the previous chromosome and can be tested for fitness. Another possible definition of a neighbour of an *n*-bit chromosome string is one that has a decimal value in one dimension (a scalar) adjacent to the previous chromosome on the number line or the *x*-axis of an (*x, y*) Cartesian coordinate graph. That is, using the first definition of a neighbour, there are exactly *n* neighbours of a chromosome with *n* bits because each neighbour is the result of changing one of the *n* bits using a bitwise OR operator.

On the *x*-axis of Cartesian 2-space, however, there are only two neighbours, one to the left that is less than the original value and one to the right that is greater. In both of these cases, there is insufficient information encoded into the chromosome to inform the genetic algorithm where a better solution is likely to be. Even in the case of only two neighbours, there is a random 50% probability that the solution is either to the left or the right and nothing in the landscape that could indicate the correct choice is visible to the GA at the current solution.

As seen previously in the knapsack problem with a solution provided by the chromosome [0111001100] = 460_{10} the closest decimal valued chromosomes (459 and 461) would return loaded knapsack values of 90 and 118 respectively, with both exceeding the maximum weight constraint. This pair's neighbouring chromosomes, further from the optimal, would yield 82 and 132, both again exceeding the weight constraint, so that a move from this direction would not lie on a continuous gradient towards the solution.

For the neighbours with one gene different from the optimal solution, the nearest neighbour that has a solution closest to the optimal is the chromosome with a decimal value that is 256 away from the optimal chromosome and a move based on a single bit change could just as easily find the solution yielding a value of 60 as it could the next best value of 95.

Finally, there is a set of solutions that are closer to the optimal and are not neighbours with the exception of one member. One chromosome that returns a solution with a value just one less than the optimal value and with a weight that is 3 below the maximum is just one step away from the chromosome with the maximum possible value. Other non-neighbouring

solutions that are better than the neighbours include those that yield the following results by value, weight, number of neighbouring moves respectively, {109, 22, 1}, {105, 24, 3}, {104, 24, 3}, {99, 24, 2}, {98, 22, 3}, {97, 19, 2}, {97, 24, 3}, {96, 21, 2}, {91, 23, 4}, so that an ascent or descent from neighbours to the optimal solution is replaced by greater movement than any ascent or descent algorithm.

In the ECU/features problem, the candidate solutions are each like a pile of coins placed on separate squares of a chess board and the search engine is only allowed to view them from above and without information from any shadows that they cast or any properties such as height or colour. In Figure 3-25, a chessboard sized grid is populated with cuboid towers whose height is unknown. The colours are not significant and bear no relation to the size of each tower. It is simply easier, if the towers are coloured, to visualise each one and its original position when the diagram is rotated in 3-D

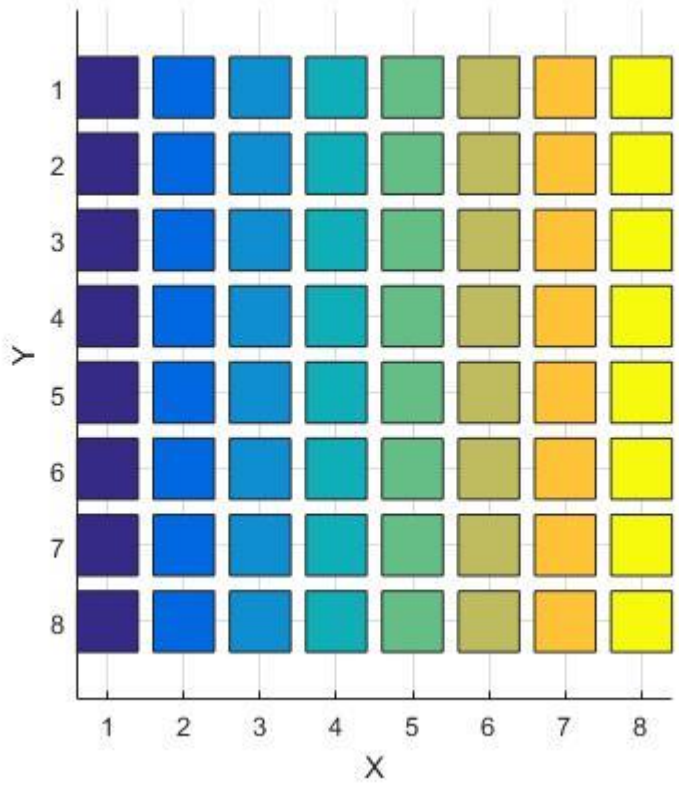


Figure 3-25 : Example of a discrete categorised search space described as a 3 dimensional landscape viewed from above

Figure 3-26 shows the same landscape, rotated and viewed from a 3D perspective, revealing the discontinuity of the data. Although it might be possible that some curve could be fitted retrospectively, there is no available function for the data that can generate the next value of Z or predict whether it will be greater or less than its immediate neighbours.

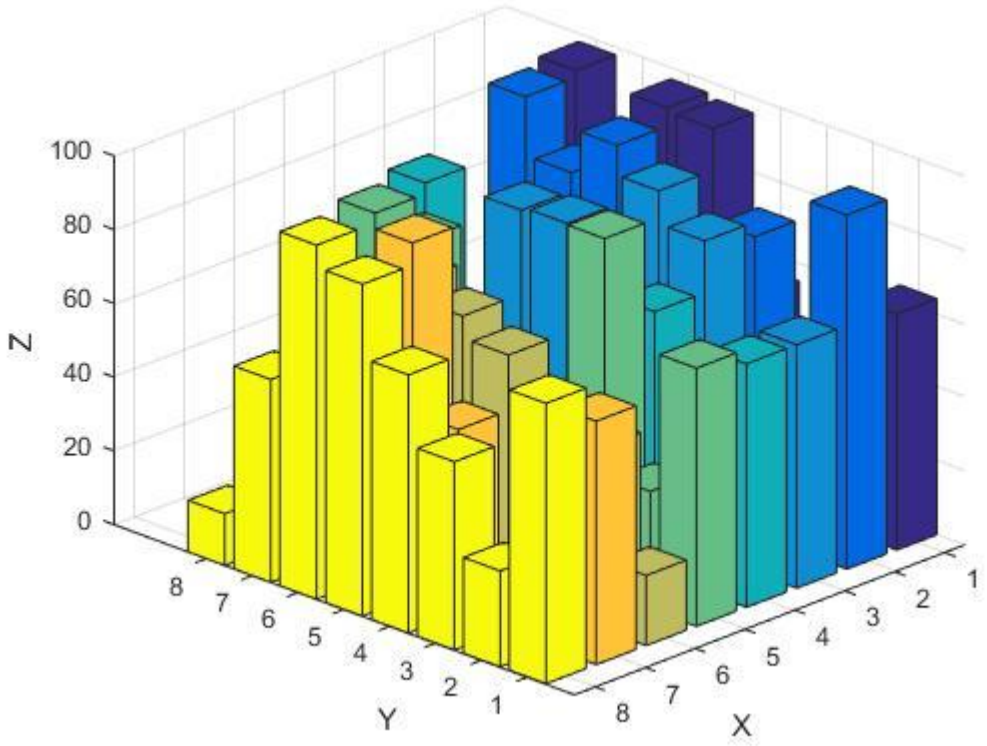


Figure 3-26 : The 3 dimensional view of the search space represented in figure 3-25

3.5.8 Software on the ECUs

Each feature in and on the vehicle will require a minimum amount of executable code which will take up a specific, finite, amount of memory on the target controller board.

This will include software for the control algorithms, drivers for the hardware and communications code for the CAN or FlexRay hubs and gateways. Whether written as bespoke packages for individual projects or taken off the shelf from well-established features such as ABS or 'Steer-by-wire', the final executable code will have a known finite size before each feature is offered to the GA as a data structure.

The chips in the controller boards will have limited capacity and will be further constrained by clock-speed, cost and other properties that will be the subject of future work, such as operating temperature range and vibration sensitivity – as well as physical size and weight.

Because modern road vehicles can have as many as 100 different features or more, each supported by ECUs, there are at least 2^{10000} unique ways that the features can be supported on any number of ECUs between a single ECU and 100 different ECUs. Searching this design space for the feasible solutions, of which there would be 100^{100} that would then be subject to further constraints, is time consuming beyond all practicalities.

One way to visualise the feasible solutions is to represent each feature and ECU combination as a set of vertical sliders. Each separate slider represents a feature that is being supported and the position of the slider represents which row of that column contains a non-zero value (the ECU that is supporting the feature).

Figure 3-27 shows a representation of one candidate solution using a system of ten sliders for a problem with ten features. The sliders can move independently in complete integer steps, thereby producing a total of 10^{10} solutions.

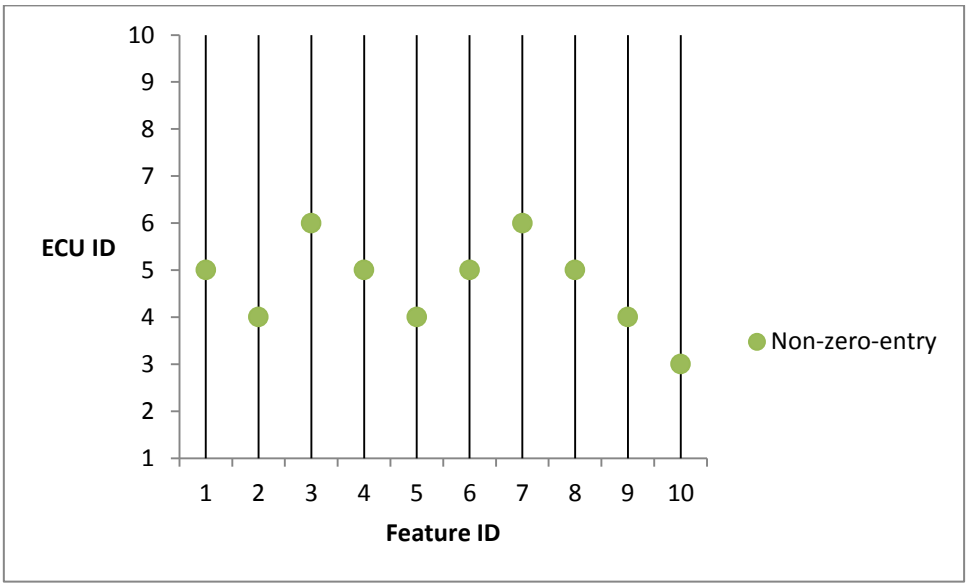


Figure 3-27 : Example of a single solution for up to ten features supported by one or many ECUs

A GA can be used to search the design space considering goals, fitness functions, linear equations and constraints. For example, in the ABS case study, there will be between one and five ECUs to manage the brake pedal, the calliper actuators, the wheel-speed sensors, the IMU sensor and the ABS controller.

In the case of just one ECU running a complete ABS, it will need a greater capacity to include the code for five features and components that can all send messages on the communication bus at a given speed, frequency and rate of transfer.

As well as sharing multiple features on a single ECU, candidate solutions should also include the scenario of the software code for a single feature being shared across multiple ECUs. This creates a matrix-management model as shown in figure 3-28, in which there is a many-to-many or “HasAndBelongsToMany” (HABTM) relationship.

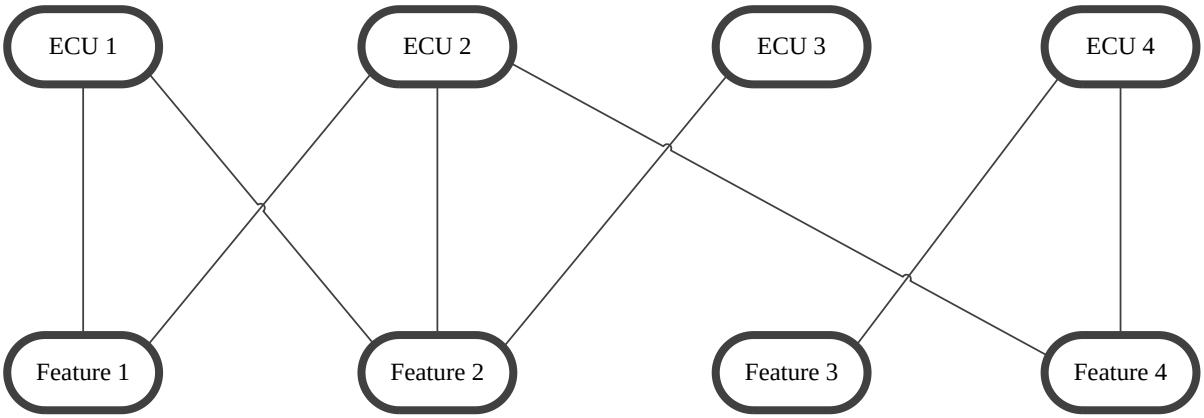


Figure 3-28 : Example of ECUs and features distributed in a HABTM relationship

In a relational database model, the many to many relationship represented in Figure 3-29 would be resolved by a junction table that is an intermediate pair of one to many relationships between ‘ECUs and ECUs/Features’ and ‘ECUs/Features and Features’ as in Figure 3-30.

Figure 3-29 : Database Entity Relation of ECUs and features

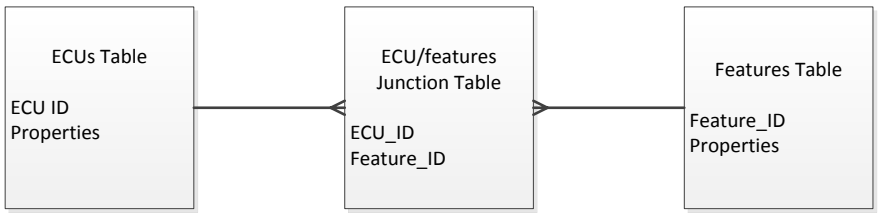


Figure 3-30 : Many to Many relationship resolved by intermediate junction table

The relationships shown in figures 3-26, 3-27 & 3-28 are represented by the square matrix where a non-zero entry specifies the pairing of ECU_ID and Feature_ID by row and column.

3.6 Results & Discussion

An initial genetic algorithm was created to generate candidate solutions for a filtering problem, mapping suitable ECUs to the features of the vehicle according to spare memory capacity and the required memory for each feature with a single memory variable.

In this model, successful candidates are those where the total memory requirement of the system is catered for across one or more ECUs with sharing of one ECU supporting multiple features being allowed.

The filter was executed once with a random generation of four chromosomes, as would be the case when the full GA is run with selection, mutation and crossover. If no suitable candidate was found, that is if any fitness vector was seen to contain a zero by manual inspection, then the model was executed again with the random seed reset so that the system therefore had no memory of the previous results. In this model, the features and ECUs were numbered starting at 1 so that a zero in the fitness vector meant that no ECU selection had been made for a given feature. Later versions of the GA numbered the features and ECUs from zero to fit with the convention of array subscripting in the 'C' programming language. Although it was not done at the earliest stages of this research, an automated inspection of the zero elements in the fitness vector could have been programmed into the model for the initial executions.

When a suitable candidate was found, for as many times as the model needed to be executed to achieve this, the number of iterations needed was recorded manually along with the successful chromosome arrangement. Again, some of this process could have been automated to save time in developing the GA, although it would have no effect on the execution times of the final implementation of the GA which did in fact incorporate this level of automation.

On initial execution of the first prototype of the GA for the ECU problem, a result after a dozen iterations produced the random chromosome [3 4 0 1 4 2] which satisfies the constraints on memory capacity for six ECUs supporting six features. This suggested that an automated run of 12 iterations of the filtering algorithm could produce a result where each feature was assigned to exactly one ECU only and that many of the runs would produce an infeasible solution where either one or more features were not supported or where a feature was assigned to two different ECUs at the same time. In table 3-11, feature zero is supported on ECU number 3, feature 1 on ECU number 4 and so on. Table 3-12 shows the same mappings exactly as they are represented by the binary-valued chromosome's 5x5 matrix, transposed to show ECUs as column headings and clearly showing ECU 5 is not used.

Table 3-11 : Table of detailed mappings between ECUs and features showing available memory versus required memory

Feature	Name	Memory	Mapping		ECU	Name	Memory Available	Memory Required	Spare Capacity
0	Brake Pedal and ABS controller	2048	3		0	Arduino	512	512	0
1	IMU and wheel speed collector	1280	4		1	MCP2515	1024	512	512
2	FL speed and Calliper	512	0		2	CAN	512	512	0
3	FR speed and Calliper	512	1		3	Intel	4096	2048	2,048
4	BL speed and Calliper	512	4		4	STM	2048	1792	256
5	BR speed and Calliper	512	2		5	ARM 3	1024	0	1,024

Table 3-12 : Simple mapping of ECUs to Features showing which ECU supports which feature

Feature \ ECU	0	1	2	3	4	5
0	0	0	0	1	0	0
1	0	0	0	0	1	0
2	1	0	0	0	0	0
3	0	1	0	0	0	0
4	0	0	0	0	1	0
5	0	0	1	0	0	0

After a further 30 trials, the number of executions for a successful candidate to be found was on average 5.57 with a 4.8 standard deviation. The maximum and minimum numbers of iterations were 18 and 1 respectively. On the four occasions when more than one suitable chromosome was generated, in the same execution, exactly two suitable but discrete chromosomes where generated – a rate of once every 7.5 executions on average. There was no occasion on which more than two suitable candidates were generated in the same run. Test scenarios were created for systems of features and ECUs that had memory requirements between 512kB and 4096kB and available memory capacity of between 512kB and 8192kB so that the ESP, the GA and the bin-packing algorithms could be executed and compared by efficiency of solution (number of ECUs required in each candidate solution) and timings (length of time taken or number of iterations required to produce a best solution).

For example, table 3-13 and table 3-14 are sections of a larger spreadsheet in which the data and results for 24 different test scenarios were recorded against the data used by the GA, the ESP and the bin packing algorithms and shows the values of the requirements and capacities of the features and ECUs respectively in arbitrary units, with the results being the candidate solutions offered by the ESP, the GA and the bin packing algorithm.

The headings of ‘ROM’, ‘Com’ and ‘Proc’ relate to ‘program memory storage’ (in ROM), ‘communication speed’ and ‘processor speed’ for verification purposes only. The attributes and requirements used in the design of the Arduino HILs model are not all the same as in this test scenario but the overall functionality of the GA is unchanged in that it maps features with given requirements to ECUs with relevant attributes and tests to find if constraints are broken or not before finding a fitness score based on how many ECUs are used and the total cost of those ECUs. Features and ECUs share common ID values since they are simply a number assigned to each individual ECU or feature of which there are equal numbers of both.

Table 3-13 : Scenario 1, an initial test scenario for the GA with 3 features and 3 ECUs

Scenario	Feature/	FEATURES				ECUs		
features	ECU ID	ROM	Com	Proc		ROM	Com	Proc
1	0	512	512	512		1536	1536	1024
	1	512	512	512		1536	1536	1536
	2	512	512	512		1536	1024	1536
3	Time	0.000009	Iterations	27				

Table 3-14 : Solution for scenario 1 showing which ECUs supported the three features

Solutions							
Exhaustive	Feasible		GA				Bin Pack
ECU 0			0:2	0:1			
ECU 1	0:1:2		1		0:1:2		0:1:2
ECU 2				2			
512	27					8	
Percentages			33%	13%	14%		(9%)

The next modification of the GA was to automate the multiple executions and to use a genetic evolution with selection of the best fitting offspring chromosome as a parent for the next generation. In this way, the system has memory of the previous generation and is less prone to retrograde mutations which cause future offspring to be weaker or less suitable. This increases the speed at which the chromosomes converge to a suitable candidate and reduces the number of iterations required on average to find a solution.

A loop was created in the code to iterate the selection, crossover and mutation of the offspring chromosomes and was set to a maximum of 30 iterations. The criterion for stopping execution before the maximum number of iterations was if any chromosome produced a pairing of ECUs and features that did not exceed the maximum memory capacity of any ECU. Therefore the GA is finding the first suitable chromosome that generates a fitness score consistent with a feasible solution, whether or not it is optimal.

Table 3-15 shows the results of running the genetic algorithm to solve a problem of assigning up to 5 features for the exhaustive model and 6 features for the reduced GA. For any of this family of problems, the number of ECUs is, initially, always the same as the number of features so that a 1:1 mapping can be chosen as the ultimate solution if appropriate.

Iterations of the executable model may or may not produce a candidate solution that uses fewer ECUs and it is possible that a solution using only one ECU is produced. The results in table 3-14 show that, for the GA, there is a linear trend with an equation

$$y = 50.357x - 28.329 \qquad (3\text{-}14)$$

$$R^2 = 0.9716 \qquad (3\text{-}15)$$

where *x* is the number of features and *y* is the time in CPU clock ticks calculated at run time. This straight line can be used to extrapolate to greater numbers of features.

Table 3-15 : Average times for execution of Features/ECUs problem

No of Features	Average times for the GA (clock ticks)	Average times for the GA (seconds)	Average times for Exhaustive Model (seconds)
2	79.33333	0.079	1
3	127.7143	0.127	2
4	158.7	0.158	29.8
5	209.5556	0.209	6758.1
6	290.2	0.290	~

Figure 3-31 shows the data from Table 3-15 as a correlation between the number of features and the execution time for the GA. Because of the exponential increase in times taken for the exhaustive model that calculates every possible candidate solution, it is not reasonable to attempt to show these on the same scale as the results for the GA.

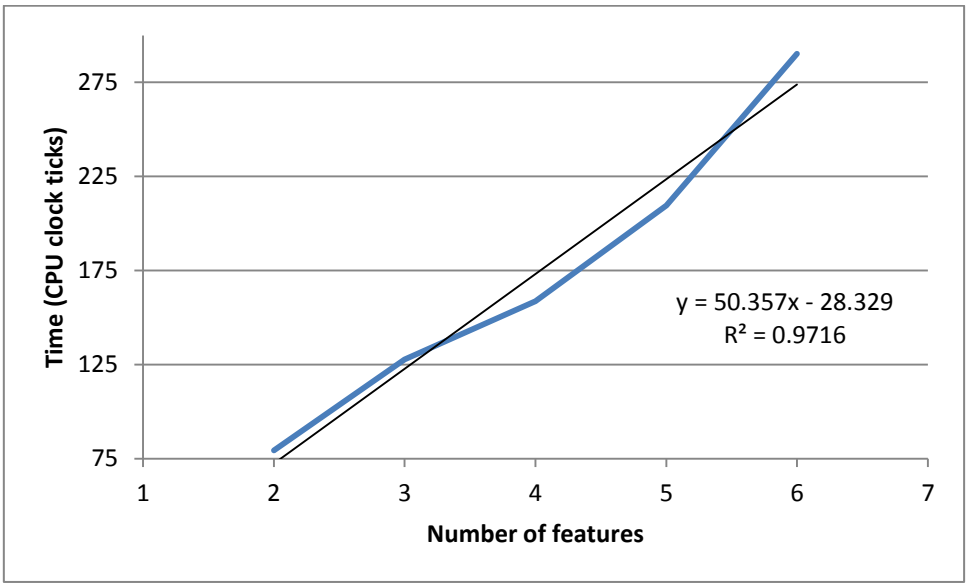


Figure 3-31 : Average times for execution of GA with systems of different numbers of features

Table 3-16 shows the average number of iterations to reach an optimal solution for both the GA (searching a subset of the solution space) and the exhaustive model (searching the entire space for every feasible chromosome).

Table 3-16 : Performance of GA compared with an exhaustive model

No of Features	Average Iterations (GA)	Average Iterations (Exhaustive)
2	1	2
3	1	6
4	4	283
5	9.5	1207
6	25.7	~

Clearly, by examination of Table 3-16, the slope (the rate of change of iteration-count with respect to the number of features) is increasing with the number of features. In Table 3-17, the results for the complete run of all exhaustive searches for two to six features are included. In these early experiments, the data is designed so as to produce a solution that will use only one ECU and the objective is to place all features on as few ECUs as possible. Each ECU is different in at least one of its memory capacities or processing or communication speeds and the result of attempting to fit all features on any ECU is reflected in the three values to the left of the features' ID numbers.

For example, in the case of six features indicated by the 6 in cell A2 (shown larger in Table 3-18, which displays the first 8 rows and six columns of the data in Table 3-17), the single ECU (ECU number 0 as indicated by the cell B2) cannot accommodate all of the features. It is deficient in communication speed (F2) for the sum of the features' requirements (F3 to F8) requiring an extra 1450 units (arbitrary units displayed as a negative value in B5) and this sets the flag in A3 to 'X' showing unsuitability.

The total ROM available on the ECU (D2 = 80000) is sufficient to support the total ROM required by the features (D3 to D8) as is the speed of processing (E2) whose overhead is shown in cell B4. With these results to hand, it is a manual process to select the ECUs that are suitable by virtue of their spare capacities after fitting the features.

Table 3-17 : Results of exhaustive search for up to six features

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD
1	Feats	ECU	Feat	ROM	Speed	Com	Feats	ECU	Feat	ROM	Speed	Com	Feats	ECU	Feat	ROM	Speed	Com	Feats	ECU	Feat	ROM	Speed	Com	Feats	ECU	Feat	ROM	Speed	Com
2	6	0		80000	30000000	1300	5	0		80000	30000000	1300	4	0		80000	30000000	1300	3	0		80000	30000000	1300	2	0		80000	30000000	1300
3	X	73856	0	1024	1000	250	X	74880	0	1024	1000	250	X	75904	0	1024	1000	250	1	76928	0	1024	1000	250	1	77952	0	1024	1000	250
4		29994000	1	1024	1000	500		29995000	1	1024	1000	500		29996000	1	1024	1000	500		29997000	1	1024	1000	500		29998000	1	1024	1000	500
5		-1450	2	1024	1000	500		-950	2	1024	1000	500		-450	2	1024	1000	500		50	2	1024	1000	500		550	2	1024	1000	500
6			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
7			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
8			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
9	6	1		800000	30000200	12590		1		800000	30000200	12590		1		800000	30000200	12590		1		800000	30000200	12590		1		800000	30000200	12590
10	1	793856	0	1024	1000	250	1	794880	0	1024	1000	250	1	795904	0	1024	1000	250	1	796928	0	1024	1000	250	1	797952	0	1024	1000	250
11		29994200	1	1024	1000	500		29995200	1	1024	1000	500		29996200	1	1024	1000	500		29997200	1	1024	1000	500		29998200	1	1024	1000	500
12		9840	2	1024	1000	500		10340	2	1024	1000	500		10840	2	1024	1000	500		11340	2	1024	1000	500		11840	2	1024	1000	500
13			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
14			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
15			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
16		2		409600	15000000	5000		2		409600	15000000	5000		2		409600	15000000	5000		2		409600	15000000	5000		2		409600	15000000	5000
17	1	403456	0	1024	1000	250	1	404480	0	1024	1000	250	1	405504	0	1024	1000	250	1	406528	0	1024	1000	250	1	407552	0	1024	1000	250
18		14994000	1	1024	1000	500		14995000	1	1024	1000	500		14996000	1	1024	1000	500		14997000	1	1024	1000	500		14998000	1	1024	1000	500
19		2250	2	1024	1000	500		2750	2	1024	1000	500		3250	2	1024	1000	500		3750	2	1024	1000	500		4250	2	1024	1000	500
20			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
21			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
22			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
23		3		409600	12000000	5000		3		409600	12000000	5000		3		409600	12000000	5000		3		409600	12000000	5000		3		409600	12000000	5000
24	1	403456	0	1024	1000	250	1	404480	0	1024	1000	250	1	405504	0	1024	1000	250	1	406528	0	1024	1000	250	1	407552	0	1024	1000	250
25		11994000	1	1024	1000	500		11995000	1	1024	1000	500		11996000	1	1024	1000	500		11997000	1	1024	1000	500		11998000	1	1024	1000	500
26		2250	2	1024	1000	500		2750	2	1024	1000	500		3250	2	1024	1000	500		3750	2	1024	1000	500		4250	2	1024	1000	500
27			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
28			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
29			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
30		4		409600	10000000	5000		4		409600	10000000	5000		4		409600	10000000	5000		4		409600	10000000	5000		4		409600	10000000	5000
31	1	403456	0	1024	1000	250	1	404480	0	1024	1000	250	1	405504	0	1024	1000	250	1	406528	0	1024	1000	250	1	407552	0	1024	1000	250
32		9994000	1	1024	1000	500		9995000	1	1024	1000	500		9996000	1	1024	1000	500		9997000	1	1024	1000	500		9998000	1	1024	1000	500
33		2250	2	1024	1000	500		2750	2	1024	1000	500		3250	2	1024	1000	500		3750	2	1024	1000	500		4250	2	1024	1000	500
34			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
35			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
36			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
37		5		409600	10000000	5000		5		409600	10000000	5000		5		409600	10000000	5000		5		409600	10000000	5000		5		409600	10000000	5000
38	1	403456	0	1024	1000	250	1	404480	0	1024	1000	250	1	405504	0	1024	1000	250	1	406528	0	1024	1000	250	1	407552	0	1024	1000	250
39		9994000	1	1024	1000	500		9995000	1	1024	1000	500		9996000	1	1024	1000	500		9997000	1	1024	1000	500		9998000	1	1024	1000	500
40		2250	2	1024	1000	500		2750	2	1024	1000	500		3250	2	1024	1000	500		3750	2	1024	1000	500		4250	2	1024	1000	500
41			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
42			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
43			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500

Table 3-18 : Results for six features and ECUs on the first ECU (ECU[0])

	A	B	C	D	E	F
1	Feats	ECU	Feat	ROM	Speed	Com
2	6	0		80000	30000000	1300
3	X	73856	0	1024	1000	250
4		29994000	1	1024	1000	500
5		-1450	2	1024	1000	500
6			3	1024	1000	500
7			4	1024	1000	500
8			5	1024	1000	500

In Table 3-19, it is shown that all of the features can be supported on the single ECU (ECU number 1 indicated in cell B9) and the supplementary 12590 units in the communication speed cell (F9) is sufficient to support the required communication speeds. The tenfold increase in ROM capacity of the ECU (D9 = 800000) accommodates the ROM requirement with plenty of spare capacity. All three criteria are satisfied and the flag in cell D10 is set to 1 to show this.

Table 3-19 : Results for six features and ECUs on the first ECU (ECU[0])

	A	B	C	D	E	F
1	Feats	ECU	Feat	ROM	Speed	Com
9	6	1		800000	30000200	12590
10	1	793856	0	1024	1000	250
11		29994200	1	1024	1000	500
12		9840	2	1024	1000	500
13			3	1024	1000	500
14			4	1024	1000	500
15			5	1024	1000	500

The data used for the runs shown in the above figures is shown in Table 3-20. Whilst these ECUs are not ordered on any of their attributes, for the purposes of the bin packing algorithm with the objective of determining a solution that allows the use of more than one ECU, there would be some way of ordering the attributes, manually before execution or automatically as part of the algorithm.

Table 3-20 : Data encoded in the ‘C’ source file of GA program

ID	Name	RAM size	ROM size	EEPROM size	Com protocol	Com speed	Com chip	Proc speed	Cost
0	"ECU_0"	4096	80000	4096	"CAN"	1300	"MCP2515"	30000000	5
1	"ECU_1"	4096	800000	4096	"CAN"	12590	"MCP2515"	30000200	4
2	"ECU_2"	4096	409600	4096	"CAN"	5000	"MCP2515"	15000000	4
3	"ECU_3"	4096	409600	4096	"CAN"	5000	"MCP2515"	12000000	6
4	"ECU_4"	4096	409600	4096	"CAN"	5000	"MCP2515"	10000000	5
5	"ECU_5"	4096	409600	4096	"CAN"	5000	"MCP2515"	10000000	6

Whilst these are not necessarily realistic values and have been chosen arbitrarily, they serve to verify the executable model. That is, to show that the GA performs as expected and as coded, values for RAM, ROM and processor speed have been chosen such that a particular solution will be provoked. These are simply possible scenarios which the GA might encounter and are a way of including input parameters that could potentially have large numbers that would encourage a solution on a single processor, for example. One might argue that these are not relevant to real world data but the point of these arbitrarily chosen values is that they test the GA and its ability to perform with a large range of data that may or may not be realistic. One must consider the distinction between verification (that the programmed algorithm performs as the coder expects it to and with respect to how it was written and the intentions of the programmer) and validation (that the model conforms to a requirement specified either by the real world or by stakeholders with an interest in the behaviour of the model - but which is not a consideration at this testing and verification stage). When the GA is validated, later, values chosen for the attributes of each ECU are real world values obtained from datasheets and from the output of source code compilation for the actual microcontrollers used in the final implementation of the validating hardware/software model.

The exhaustive runs show the correct optimal solution as using a single ECU in all cases, usually either ECU_0 for fewer features with lesser requirements and ECU_1 for larger numbers of features or with greater requirements. Whilst these do not match the solutions obtained by early runs of the non-exhaustive GA, they give an indication of the optimal solution towards and upon which the GA should be converging.

Altering the values for each of the variables produces different solutions on one or more ECUs with one or more features supported on each. There were three attributes originally considered for the model, ROM size, processor speed and communication speed. These were initially chosen because they were considered to be potentially important requirements of the system but because of the use of Arduino microcontrollers later in the research and for the case studies discussed in chapter 4 and chapter 5, other attributes of the ECUs were easier to obtain from datasheets and compilation output that gave clearer indications of both the requirements of the features and the capacities of the microcontrollers. The pertinent variables are ‘program storage space’, ‘local variable space’ and ‘processor speed’ which were all known for the Arduino microcontroller boards and from the code compiled for each feature. Further work, discussed later in this chapter highlights other variables that could be considered in future variants of the executable models, so that use of the GA is not restricted only to just one type of microcontroller or to their attributes chosen for the case studies in this research. The GA is flexible, scalable, reusable and adaptable to many different ECU/features and hardware/software allocation problems. For verification of the GA, the experiments could theoretically use any variables defined by the features’ requirements and supported by the same properties of the ECUs as long as they matched in number the variables in the evaluation function. Using just three variables, the ESP was executed to obtain an optimal

solution by the definition of the fitness score and this solution was then compared with candidate solutions from the GA.

The ESP was executed overnight for the 6x6 case and ran for five hours. The algorithm produced every possible chromosome exhaustively from $0 \leq y \leq \left(2^{n^2} = 68{,}719{,}476{,}736\right)$ and verified that the expected number of feasible chromosomes, $n^n = 6^6 = 46656$ were identified during the program execution and passed as valid and feasible chromosomes according to the algorithm's checks. The refinement of the algorithm so as to generate only the feasible chromosomes and dispense with the need to check the feasibility of chromosomes as they were generated reduced the total number of iterations for the ESP to just the 46656 feasible chromosomes, reducing the runtime for all subsequent models, whatever the number of features and ECUs.

Two factors that greatly influence the number of candidate solutions found and the percentage of occasions on which any given solution is offered are

a) the maximum number of loops that the GA can perform and

b) the number of times the GA itself is run

The size of the problem introduces other effects that can cause the GA to perform less well when fewer features are being considered. For the saving in time for models with fewer than five features, no advantage was found in using the random GA over the ESP. For the specific case of a four-feature system, table 3-21 shows results after increasing the number of runs and/or the number of maximum possible iterations per run where the maximum possible solutions is $4^4 = 256$.

Table 3-21 : Requirements and attributes of the features and ECUs for an example four-feature system

Feature	ROM	Proc	Com		ECU	ROM	Proc	Com
0	4096	4096	4096		0	4096	4096	4096
1	2048	2048	2048		1	4096	4096	2048
2	1024	1024	1024		2	2048	2048	2048
3	512	512	512		3	1024	1024	512

The GA uses a counter based on powers of two and, to perform this many loops, the power of two required is $2^8 = 256$, therefore a value of 8 is passed to the code at runtime or before compilation. However, to perform this many iterations, the GA is a wasteful resource and it only makes sense to use the GA if fewer than 256 iterations are performed for this specific problem. As a guide, the GA should only be run with no more than half of the iterations that would be needed for the ESP. Since the mechanism for setting the number of loops is by using a power of 2, the next size possible with integers is $2^7 = 128$

3.6.1 Verification of the GA and bin packing algorithms

For the specific case of six features (because this is possible to execute on both the Bin Packing and the GA programs) the following results in table 3-22 were obtained for the given data, chosen arbitrarily. It shows the allocation of ECU resources to the six unique features suggested by both the GA and the bin-packing algorithm. The same solution was discovered by both, albeit the exact same ECUs were not used in both solutions. The memory capacity of the ECUs and the features they supported were identical in the GA and the bin-packing algorithm solutions, that is single ECUs with 2048kB of memory each support feature 0, features 1 & 2 and features 3, 3 & 5. The minimum number of ECUs that can support the features is three, as verified by the ESP, although again the possibilities of which three are many for this problem because each ECU has the same capacity as the others. The ESP delivered a solution on ECUs 1, 2 & 3 with half of the features on ECU 0 and just one on ECU 2.

Table 3-22 : Results of bin-packing and GA for a problem with six features

Feature		ECU		Solution		ECU mem used	
ID	Mem rqrd	ID	Mem cap	GA	Bin Pack	GA	Bin Pack
0	2048	0	2048	3	0	2024	2048
1	1096	1	2048	2	1		1608
2	512	2	2048	2	1	1608	2024
3	512	3	2048	0	2	2048	
4	1000	4	2048	0	2		
5	512	5	2048	0	2		
Total	5680		12288			5680	5680

The GA performed well with only 512 iterations (the total feasible search space is 46656 chromosomes) before it converged on the same (deterministic) solution obtained from the bin-packing algorithm. The maximum number of iterations for the bin-packing algorithm is

$$n(n+1)/2 \qquad (3\text{-}16)$$

or the triangular number of *n* and, whilst this is less than the total number of feasible candidate solutions from the GA, the bin-packing algorithm will usually perform only few of the many coded iterative steps (loops) before finding a suitable ECU and breaking out of the loop. This greatly reduces the number of iterations to less than 2n-1 and even further below n^n, with a theoretical minimum number of iterations being just *n*, in the case where all of the features would fit on to the first ECU.

The trade off, for the speed of the bin packing algorithm, is that it can only find one solution and it is likely to be sub-optimal, considering the possible solutions from which it chooses the one it does.

A combination of the GA and bin-packing algorithms, executed in parallel, allows for a comparison between the two to test whether one produces a better solution than the other. They can also be assigned to specific tasks within the ECU design problem, that is the software/hardware architecture of each localised network can be solved with a bin-packing algorithm while the GA can be employed to solve the problem of communication between clusters of networks and the testing, verification and validation of the communication protocols deployed.

3.6.2 Finding an optimal number of iterations for the GA

Normally, a GA will stop executing either when a predetermined number of offspring have been generated and tested for selection, or when the generation of chromosomes converges according to some arbitrary measure, often the number of times the same solution is found or when all of the chromosomes in any new generation are exactly the same.

One observation from the results is that for runs of the GA with a higher number of iterations, generations or offspring/mutations/crossovers, some previously seen candidate solutions disappear and the number of candidate solutions found and suggested by the software decreases. When far fewer iterations are allowed, there is a higher likelihood that the default solution of one ECU per feature is returned by the GA. The ideal is to allow sufficient iterations that the weaker chromosomes do not survive but not so many that the number of iterations or the execution time of the GA approaches that of the exhaustive model – even the exhaustive model with the reduced n^n possible chromosomes.

In the case of six ECUs/features, raising the maximum number of iterations from 4096 to 16384 produced a reduction in the number of candidate solutions offered from seven to just three. This is still a 30% reduction in the number of iterations compared with the reduced exhaustive model.

3.6.3 Results of changing the order of items in the bin packing algorithm

Table 3-23 shows the results of test scenarios 18 and 19 (full table in Appendix E) where the order of the items in the data structure were not ordered (scenario 18) and then ordered by ROM size (scenario 19). Both solutions involve 4 ECUs but the number features assigned to each is not the same in any two solutions. For the bin backing algorithm, although deterministic, the order in which the items have been presented has generated a different solution because no ordering of the items is performed during execution of the algorithm. Similarly, the ESP has generated two different candidate solutions for the same data, because of the order in which the items were presented. When the optimum solution is unique in terms of the number of ECUs, or the cost of using the same number of ECUs in a different way is different, the ESP will not find more than one solution. However, if there are many solutions that have the same cost and use the same number of ECUs, the ESP will find them all but will not replace a solution later in the execution if its fitness score is not better. In this way, it can also make a difference to the ESP in the same way as the bin packing algorithm if the data is presented in size order or not.

Table 3-23: results of bin packing and ESP for 8 features with and without ordering

		FEATURES			ECU	Bin	Optimum	Candidates
Scenario	Features	ROM	Com	Proc	ID	Packing	found?	Exhaustive
18	0	4096	4096	4096	0	1:3:4:5	4 ECUs	3:4:5:6:7
	1	2048	2048	2048	1	6:7		2
	2	1024	1024	1024	2			1
	3	512	512	512	3			
	4	512	512	512	4			
	5	512	512	512	5			
	6	1024	500	1000	6	0		0
	7	1024	500	1000	7	2		
8	Time	0.08253	Iterations		18trn			16m
		FEATURES			ECU	Bin	Optimum	Candidates
Scenario	Features	ROM	Com	Proc	ID	Packing	found?	Exhaustive
19	0	4096	4096	4096	0	2	4 ECUs	6:7
	1	2048	2048	2048	1	0		2:3:4:5
	2	1024	1024	1024	2	1:3:4		0
	3	1024	500	1000	3	5:6:7		1
	4	1024	500	1000	4			
	5	512	512	512	5			
	6	512	512	512	6			
	7	512	512	512	7			
8	Time	0.08253	Iterations		18trn			16m

Whilst the GA searches the design space in a random fashion, the bin packing algorithm is deterministic and will always find the same solution to any given problem no matter how many times it is run. The bin packing algorithm is also a function of the order in which the bins and items are examined and this can ultimately affect the possibility of ever finding the global optimum solution for some problems.

3.6.4 Average number of iterations before convergence

The GA results show that for larger problems, more iterations are required, on average, for each new chromosome, before the GA ceases offering any more solutions. That is, for a system of only four features, a maximum of four solutions are offered in any single run of the GA and they occur more than 90% of the time in the first 52 iterations (chromosomes) out of 3125. Similarly, for a system of eight features, the solutions are all found within the first 50 iterations. For any number of features, n, the total number of possible distinct chromosomes, whether feasible or not is 2^{n^2} and the total number of feasible chromosomes that reasonably map every feature to at least one ECU each is n^n. The ESP will perform exactly 2^{n^2} iterations to exhaustively examine every possible chromosome and find the optimal solution.

To determine the maximum number of iterations, *i*, that the GA will perform, the log (base 2) of the number of feasible chromosomes is taken and the child chromosomes are generated up to a maximum of 2^l times where $l = log_2(n^n)$. In some cases, where the value of n^n is an exact power of 2, that is $n^n = 2^k \; where \; k \in \{\mathbb{N}\}, \; then \; 2^l = n^n$ and the GA would perform

as many iterations as the ESP. When this happens, the GA should be performed 2^{l-1} times which will produce exactly half as many chromosome tests as the ESP and therefore perform only half as many iterations.

3.6.5 Performance of the GA

After analysing the average number of iterations that the GA would take before reaching the last useful replacement of the parent chromosome with a better offspring chromosome, executions of the GA with limits on the maximum number of iterations allowed between new solutions and a total maximum limit on the number of iterations overall, were performed. Positive results showed that the GA didn't need to run for as long as predicted from the ESP data, because no more useful solutions were found after less than half of the program had executed. In the extreme, a GA searching for a solution in a system of 99 features, could find useful solutions after just 4 seconds compared with the actual time of over six hours for the ESP with only 10 features or the predicted time of $10^{24.3952}$seconds or 5.7 x 10^6 x age of the universe.

Synthetic data were created for the features and ECUs so that the GA could be used to solve a system of 12 features with a known solution calculable by hand. The features were assigned identical attributes across all twelve and the ECUs were such that three distinct configurations of attributes existed and were shared by at least three ECUs. The precise configuration of features and ECUs was such that clearly by inspection there were sufficient ECUs to support all of the features on just three ECUs with sufficient capacities to support four features each. This meant that the GA too should have been able to find a solution on three ECUs with four features on each.

The ECU and features data are shown in Appendix F and Appendix G exactly as they appear in the source file of the 'C' program for the GA. Data is written into the variable declarations of the code rather than read from a file, although the latter method is suggested for further work as a method of increasing reusability of the code. From 100s of executions, the GA was unable to find a better solution than on four ECUs and, whilst this was not as good as the bin packing algorithm, it was much better than the default of one feature to one ECU, saving on eight ECUs.

With the data sorted so that the ECUs with the largest capacity came first in the list of ECU data, a single run of the GA set to a maximum of 4096 iterations of which 461 were used in a time of 0.089 seconds offered a solution allocating three ECUs each supporting four features. Although different features were shared from the ones allocated to each ECU in the bin packing problem, this is a valid solution on the basis that the algorithm is simply attempting to minimise the number of ECUs used and it has matched the bin packing algorithm and the ESP to give

ECU 0 supports : 0 : 8 : 10 : 11
ECU 2 supports : 2 : 3 : 7 : 9
ECU 4 supports : 1 : 4 : 5 : 6

Subsequent runs of the GA with the data in this sorted order did not yield a solution on three ECUs. An output log file from a single execution of the ESP model with an upper limit of 12^{12} iterations is included in Appendix H.

The solution offered is the predicted three ECUs of minimum cost. No units are suggested or implied in the numerical value of cost. The maximum number of allowed iterations in this single execution is 39091130 and the solution array represents the binary valued vector 000011112222 since one of the arrangements of the 12 possible chromosomes needed to place four features on each of three ECUs is

$$\begin{bmatrix} 000000001111 \\ 000011110000 \\ 111100000000 \\ 000000000000 \\ 000000000000 \\ 000000000000 \\ 000000000000 \\ 000000000000 \\ 000000000000 \\ 000000000000 \\ 000000000000 \\ 000000000000 \end{bmatrix}$$

meaning that the first four features are supported on ECU 0, the next four on ECU 1 and the final four on ECU 2. The non-zero values in each of the first three rows of the matrix represent four ‘on-switches’ in each of the placeholder values for 0, 1 and 2

The number represented by the matrix is 000011112222 in base 12 or just 11112222_{12} which is 39091130_{10}

Therefore the total value of the non-zero elements is

$$12^7 + 12^6 + 12^5 + 12^4 + 2\text{x}12^3 + 2\text{x}12^2 + 2\text{x}12^1 + 2\text{x}12^0 \qquad (3\text{-}17)$$

The matrix is reversed with respect to the usual order of placeholder values for ease of programming and is easily corrected by the program before being displayed as a decimal number. Whilst the exhaustive model is restricted to 11112222_{12} iterations, this is sufficient to show that a solution with only three ECUs is possible and this is then a target for the GA to chase.

With runtimes as low as less than 0.5 seconds, the GA does not always find the deterministic optimum value but the number of iterations can be increased and still remain within acceptable levels. Whilst the optimum solution is found occasionally, it is rare. However, the GA moves very quickly away from the default solution of one ECU per feature and converges on a solution that uses only four or five ECUs on almost every run, rather than the

expected three. The GA is undoubtedly finding better solutions than the starting position and is doing so in less than half of a second. Occasionally, three ECUs are offered as candidates and sometimes, but not always, the most financially economical three.

3.6.6 Performance of the bin packing algorithm

Comparing this result with the Bin-Packing algorithm, the following output, directly from execution of the bin-packing program for the same data but without considering cost, shows a solution that places four features on each of three ECUs (the first three ordered by capacity) but with no algorithm for calculating a fitness score by ECU cost, any three ECUs with a capacity of 400 would be suitable according to this program. The data were entered into the executable in their original ID orders, with names reflecting the IDs. On the first pass of the bin packing algorithm, the data are rearranged and ordered according to their capacity scores. The new order that the bins are given is reflected only in the ID and the 'POS' values. In this example, the result of the reordering and sorting can be seen by the seemingly unordered names which, in reality, have no order other than any imposed by human prejudice. In this example, the names were created only to show the effect of the sorting by capacity scores. The sorted bins, ID, Name and POS will appear as in table 3-24 but they may move up or down the bin array. From the execution of the bin packing program, the output is shown in table 3-25 to reflect the finishing states of the used bins. Table 3-26 shows the output of the bin packing program displaying the data in the items fields.

Table 3-24 : Result of sorting bins at the start of the bin packing algorithm

ID	NAME	POS	USED	ROM	C_speed	P_speed	capacity_score
0	Zero	0	0	400	400	400	692.820313
1	One	1	0	400	400	400	692.820313
2	Two	2	0	400	400	400	692.820313
3	Six	3	0	400	400	400	692.820313
4	Seven	4	0	400	400	400	692.820313
5	Nine	5	0	360	300	300	556.417114
6	Ten	6	0	360	300	300	556.417114
7	Eleven	7	0	360	300	300	556.417114
8	Eight	8	0	312	312	312	540.399841
9	Five	9	0	312	312	312	540.399841
10	Three	10	0	312	312	312	540.399841
11	Four	11	0	312	312	312	540.399841

Table 3-25 : Redirected output after execution of bin packing algorithm

ID	NAME	POS	USED	ROM	C_speed	P_speed	capacity_score
3	Six	3	0	400	400	400	692.820313
4	Seven	4	0	400	400	400	692.820313
5	Nine	5	0	360	300	300	556.417114
6	Ten	6	0	360	300	300	556.417114
7	Eleven	7	0	360	300	300	556.417114
8	Eight	8	0	312	312	312	540.399841
9	Five	9	0	312	312	312	540.399841
10	Three	10	0	312	312	312	540.399841
11	Four	11	0	312	312	312	540.399841
0	Zero	0	1	0	0	0	0.000000
1	One	1	1	0	0	0	0.000000
2	Two	2	1	0	0	0	0.000000

Table 3-26 :Redirected output of bin packing algorithm showing items data

ID	ROM	C_speed	P_speed	bin ID
0	100	100	100	0
1	100	100	100	0
2	100	100	100	0
3	100	100	100	0
4	100	100	100	1
5	100	100	100	1
6	100	100	100	1
7	100	100	100	1
8	100	100	100	2
9	100	100	100	2
10	100	100	100	2
11	100	100	100	2

At the end of execution of the bin packing algorithm, the candidate solution is displayed as a vector of ECU numbers assigned in the order of the element numbers of the vector starting at zero. In the specific example of this redirected output of the execution

alt_out_list looks like this
[0 0 0 0 1 1 1 1 2 2 2 2]

means that the suggested solution based on bin packing is as follows

Bin 9: ID 0: name Zero supports : 0 : 1 : 2 : 3
Bin 10: ID 1: name One supports : 4 : 5 : 6 : 7
Bin 11: ID 2: name Two supports : 8 : 9 : 10 : 11

The value of the vector ‘alt_out_list’ represents the mappings of the first four features to ECU 0, the next four to ECU 1 and the final four to ECU 2

3.6.7 Summary of the GA software code

The chromosomes and genes used in the final version of the genetic algorithm were represented in the 'C' programming language as an *n x n* array of *n* row vectors, each containing *n* elements where *n* is the number of features and also the number of ECUs. That is, for example, in the case of four features and four ECUs, a single chromosome would have 16 elements arranged so as to be read as either a single vector of 16 genes or as a 4 x 4 matrix, in which the intersection of rows and columns represent the allocation of a feature to an ECU with a non-zero entry meaning that a feature has been allocated an ECU and a zero meaning that the allocation has not been assigned.

Crossover was coded to occur at the generation of every new offspring with a probability of 100%. That is, from the parent pair, two children were created by taking genes from the beginning of one parent's chromosome and splicing genes from the end of the other's. For example, considering the parent pair

a = [1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1]

b = [0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0]

which represent the matrices

$$A = \begin{bmatrix} 1\ 1\ 0\ 0 \\ 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 1 \end{bmatrix} \quad , \quad B = \begin{bmatrix} 0\ 0\ 0\ 1 \\ 1\ 0\ 0\ 0 \\ 0\ 0\ 1\ 0 \\ 0\ 1\ 0\ 0 \end{bmatrix}$$

a crossover point is selected randomly to coincide with a line drawn between two adjacent columns, for example, separating the third and fourth columns, thus

$$A = \begin{bmatrix} 1\ 1\ 0 \mid 0 \\ 0\ 0\ 1 \mid 0 \\ 0\ 0\ 0 \mid 0 \\ 0\ 0\ 0 \mid 1 \end{bmatrix} \quad , \quad B = \begin{bmatrix} 0\ 0\ 0 \mid 1 \\ 1\ 0\ 0 \mid 0 \\ 0\ 0\ 1 \mid 0 \\ 0\ 1\ 0 \mid 0 \end{bmatrix}$$

and the column(s) to the right of the line in A are swapped, or crossed over, with the corresponding column(s) in B to produce two new children, C and D

$$C = \begin{bmatrix} 1\ 1\ 0\ 1 \\ 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0 \end{bmatrix} \quad , \quad D = \begin{bmatrix} 0\ 0\ 0\ 0 \\ 1\ 0\ 0\ 0 \\ 0\ 0\ 1\ 0 \\ 0\ 1\ 0\ 1 \end{bmatrix}$$

and this is achieved in code with the following function, expressed here in pseudocode, with an indenting style similar to that used in the Python PEP 8 standard

```
crossover_function()
    // considers the column wise orientation of the chromosome in a square matrix
    create a random crossover point
    loop twice to consider the parent chromosomes
        loop for as many ECUS or rows in the system
            loop for as many features or columns in the system
                if the column is greater than the crossover point
                    write the value from parent 1 to child 2
                    write the value from parent 2 to child 1
                otherwise
                    write the value from parent 1 to child 1
                    write the value from parent 2 to child 2
```

Mutation was programmed to occur with a probability of 85% and, because of the peculiar constraints on the arrangement of non-zero values in the columns of the chromosome array, a mutation in the context of this GA was not a simple change in the value of a single gene but rather a swapping of a non-zero from one position in a column to another position in the same column. For example, a mutation in a 4 x 4 array could make a transition from A to B such as

$$A = \begin{bmatrix} \mathbf{1} & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \rightarrow \qquad B = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ \mathbf{1} & 0 & 0 & 0 \end{bmatrix}$$

representing a mutation in the first column that results in the switched-on gene in the top row being replaced by a switched-on gene in the bottom row.

This was implemented in ‘C’ with a function that used this pseudocode logic, again with a PEP 8 style for indentation

```
mutate_function()
    // causes a random mutation in child chromosomes
    loop through child chromosomes one by one
        in 10% of cases do not mutate and just leave the function
        select one row at random
        select one column at random
        set the previously non-zero gene in this column to zero
        set the gene at the selected row and column to 10%
```

The fitness function is based on the total number of features that any chromosome can support so that the best possible fitness score is one that supports every feature. Beyond that, the number of ECUs that are used to support those features is considered with less weighting than the number of features supported and finally (if the program is compiled with this functionality in mind) the total cost of the ECUs that the chromosome indicates are to be used in the solution. So that the code can be represented as the pseudocode below

if costs are not being considered
    fitness = total_features + (0.1/(1+ECU_usage));
otherwise
    fitness = total_features + (0.1/(1+ECU_usage)) + (0.0001/(1+total_cost));

where the order of magnitude of the ECU usage is reduced to 1/10 and the order of magnitude of the total cost is reduced to 1/10000 to make a single fitness score that takes account of the importance of the number of features supported, while placing less importance on minimising of number of ECUs and only considering cost if the first two are identical to a previous solution.

3.7 Conclusions

The ECU problem proved difficult because of the number of calculations needed to examine every possible combination and permutation of candidate solutions for a given number of ECUs and features in a vehicle. An exhaustive search of the entire solution space was possible for systems of up to 9 features and two distinct problems arose after this. The first was a limitation on the largest integer that the 32-bit implementation of 'C' could represent. Simply trying to code the executable program to count up to anything higher than $2^{32}$was, whilst not impossible, very difficult. It was possible to create user-defined number types with greater than 32 bit representation and the exhaustive model of the GA ('ESP') did move towards a character based counting system that populated a string with 1s and 0s in a fashion that replicated binary addition beyond the maximum value possible with just 32 bits. The "string.h" standard 'C' library function 'strncmp' was used to test the value held in the string and to check that it had reached the target value for program termination.

The second problem was much harder to address and could only be improved upon by using machines with faster processors. Even with the ability to count much higher than the unsigned 32-bit integer would normally allow the length of time taken for a program to perform that many individual processes to test every possible permutation of ECU and feature was prohibitive. Extrapolating from tables of execution times taken for systems of up to 10 features suggested that for as few as 15 features it would take 10^{10} seconds or over 3 centuries to execute the program.

This could be reduced with heavily parallelised code but even using 100 processors or 100 individual desktop machines would only reduce the time needed to 10^{8} seconds which would still be more than 38 months and this still only for 15 features.

The bin packing algorithm reduced the amount of time taken to just a few seconds, even for very large numbers of features but it only ever examined or delivered a single solution for any given problem and it was often sub optimal. The bin packing algorithm was not guaranteed to find the global optimum solution, although it did provide a starting position for the GA that was already better than the default 1:1 ECU/feature mapping when possible. The

hypotenuse normalisation of the different units of measure, scale and range of each attribute of the ECUs proved effective in discounting the bias that any individual feature had in the ordering of the bins (ECUs).

The genetic algorithm drastically reduced the number of solutions examined compared with the total number that were possible and therefore reduced the time taken to find a better solution. It could also test more solutions than the bin packing algorithm and whilst there were no guarantees of finding the optimal solution, the GA performed well enough to find better solutions than the default starting position on every execution.

Whilst the GA performs well to find better solutions in a reasonable amount of time, the use of n x n matrices to code the chromosomes means that some traditional methods of implementing GAs with a single vector for the chromosome cannot be implemented. Chromosomes that do not have to follow a strict format (unlike the ones that must have a single non-zero in each column of an n x n array) allow for crossovers and mutations that do not break any rules about the types of representations that make the chromosomes valid or invalid and do not need to be tested before they are used to find a fitness score. It would be interesting and potentially useful to investigate the use of vector chromosomes to solve the ECU problem. Attempts were made in this research to use a vector chromosome but the need to then use integer values that went beyond the binary 0-1 made them less easy to manipulate with the benefits of binary arithmetic that computerising the problem gave.

No advantage was found in using larger population sizes than one pair of parents producing two offspring in each generation. Since the populations generated by GAs do not suffer from the effects of in-breading that a biological subject might, there is no issue with incest and any benefits that larger populations provide are outweighed by the time it takes to generate them.

For future research into the use of bin packing algorithms, it would be useful to explore alternative algorithms and the hypotenuse normalisation could be improved but these were beyond the scope of this research and could be investigated as part of a different project.

4. Case Study 1 : Antilock Braking System

"Antilock braking systems (ABS) are closed-loop devices designed to prevent locking and skidding during braking" [63]. ABS is a system that monitors and analyses the speeds of the vehicle and the individual wheels while the brake pedal is being applied so that the brake demand can be modulated to prevent the wheels from locking or slipping. Braking distances can be greatly reduced by ABS in wet or slippery conditions when drivers cause skidding by harsh braking. The ABS controller intervenes and performs 'cadence braking' to allow the vehicle to come to a complete stop under control and with steering function preserved. ABS does not always reduce the stopping distance compared with skidding to a halt but the retention of steering function and the accompanying vehicle stability allows for a more controlled deceleration with the ability to move around stationary objects and other vehicles, thereby avoiding collisions or damage to vehicles, drivers, passengers and pedestrians, property or animals.

"Typical ABS components include: vehicle's physical brakes, wheel speed sensors (up to 4), an electronic control unit (ECU), brake master cylinder, a hydraulic modulator unit with pump and valves", as described in [64]. Other components of the ABS can be programmed separately and hosted on individual ECUs, from the brake pedal itself to the callipers as shown in figure 4-2.

Although this research is aware of many alterative deceleration models, such as those highlighted by Maurya [65] and hailed as being more realistic than constant deceleration models, this research is content with the use of a constant deceleration model as a verification for the ABS controller algorithm. With this model, braking behaviour can be accurately predicted for the hardware simulation and used as a reference to compare results. Even the ABS controller algorithm for this research is not meant to be an improvement on any existing system but rather is a component in demonstrating that the ABS system design (in software and hardware) in this research is valid. The intention is that at some later stage, any ABS controller and/or behavioural model(s) with relation to positive or negative acceleration can be incorporated into a vehicle model and can be swapped out for more realistic ones if possible/necessary. In a production vehicle for the road, existing ABS controller software will be available and the physics model will be replaced by a HILs model using an actual vehicle.

Because the algorithms for this ABS case study are designed to run on desktop PCs, μC boards and ultimately be deployed on ECUs within an embedded system, they need to be written in a reusable and platform independent way that can be compiled differently depending on compiler directives. This means that there are programming concerns and issues around simulating the communications protocol when individual nodes on the desktop system can directly access each other's data via shared memory and on the distributed system when two distinct features that previously had their own separate areas of memory become integrated on the same ECU and no longer need to use the CAN but to talk to each other.

Models designed as output of this research and third party hardware/software are brought together by the application of heuristic modelling software that this research has developed to allow the rapid prototyping of an ABS system for integration with other embedded electronic vehicle systems. Figure 4-1 shows the components of the ABS system and how this research provides models of ABS at various stages that inform the modelling code to allow the engineers to make informed decisions about how to integrate the bespoke software with the third party hardware/software that support the final implementation of the system.

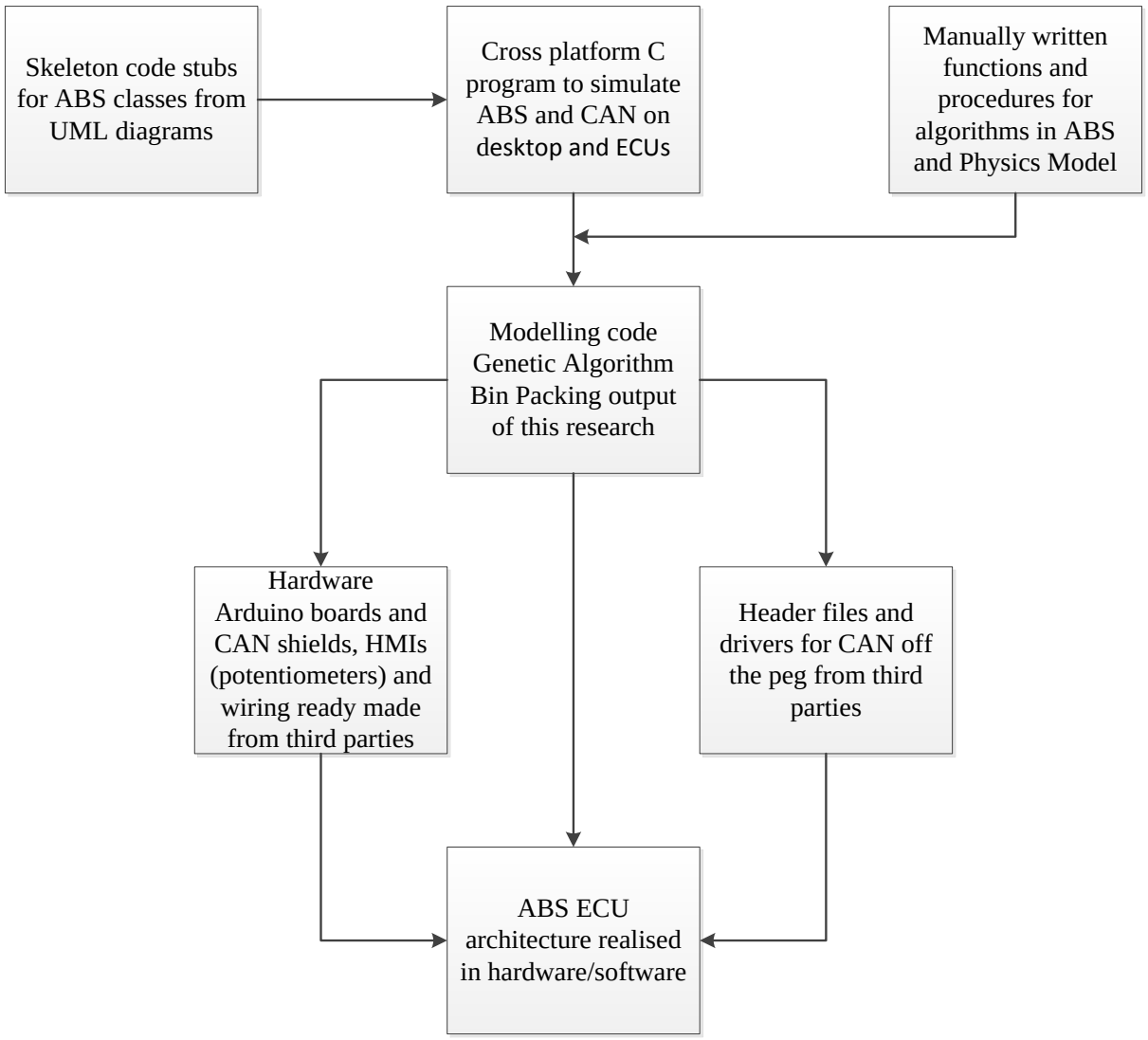


Figure 4-1 : Top level diagram of end to end process developed by this research

4.1 Components of Anti-lock Braking System (ABS)

Figure 4-2 shows the main components of a typical ABS system that this research is interested in modelling, starting with the human machine interface (HMI) that is the brake pedal. ABS is not concerned with events that occur when this is not pressed and so, as well as being the control device for the amount of brake pressure the driver wishes to apply, it is also a switch to turn on the ABS controller's monitoring functions that determine whether there is a need for ABS to intervene and activate brake modulation.

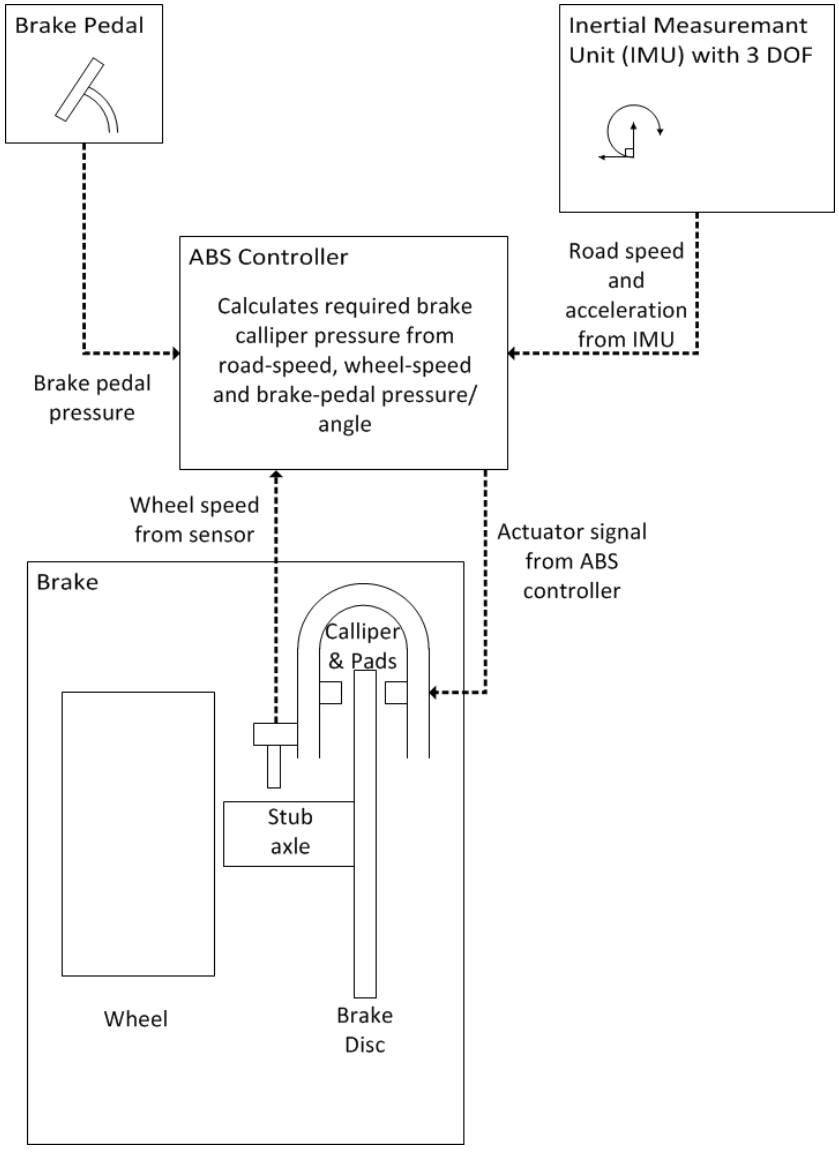


Figure 4-2: Author's representation of the components of an antilock braking system (ABS)

The controller receives sensor data from the pedal and from speed sensors at each wheel as well as from an inertial measurement unit (IMU) that senses the vehicle's road speed

independently of the wheel speeds. It is the comparison of the wheel speeds with each other and with the IMU that determines the occurrence of an ABS event.

The brake pressure demand, finally sent to the brake callipers as a braking force in the hydraulic fluid lines to the callipers, is calculated by the ABS controller and, if different from the demand requested from the HMI, a reduction in brake pressure is applied. This reduction may be as much as to completely release the brake callipers from the discs and apply zero brake pressure at one or more wheels so as to allow free rotation of the wheel back to an appropriate speed to control the vehicle or to attempt further braking.

4.2 Methods

Preliminary designs of the ABS system hardware/software architecture in this research leant on UML to create class descriptions with classes and members. Whilst one advantage of this was that it was possible to generate source code automatically from the diagrams, much of the code needed to be manually entered from the keyboard into the diagrams themselves (or behind them in an action language) and so less time was saved by this process than would have been ideal. The first, highly abstracted, model has just four elements; The ABS controller, brake pedal, an inertial measuring unit (IMU) and the four wheels.

Figure 4-3 shows the 1:1 relationships and attributes of the braking system that will be converted into stubs (skeleton source code in C/C++) to be edited by the software designer and then compiled into executable code. The ABS package diagram itself includes model class descriptions for each of ‘ABS_control’, ‘IMU’, ‘Pedal’ and ‘Wheel_Brake” (latterly “wheel(s)”).

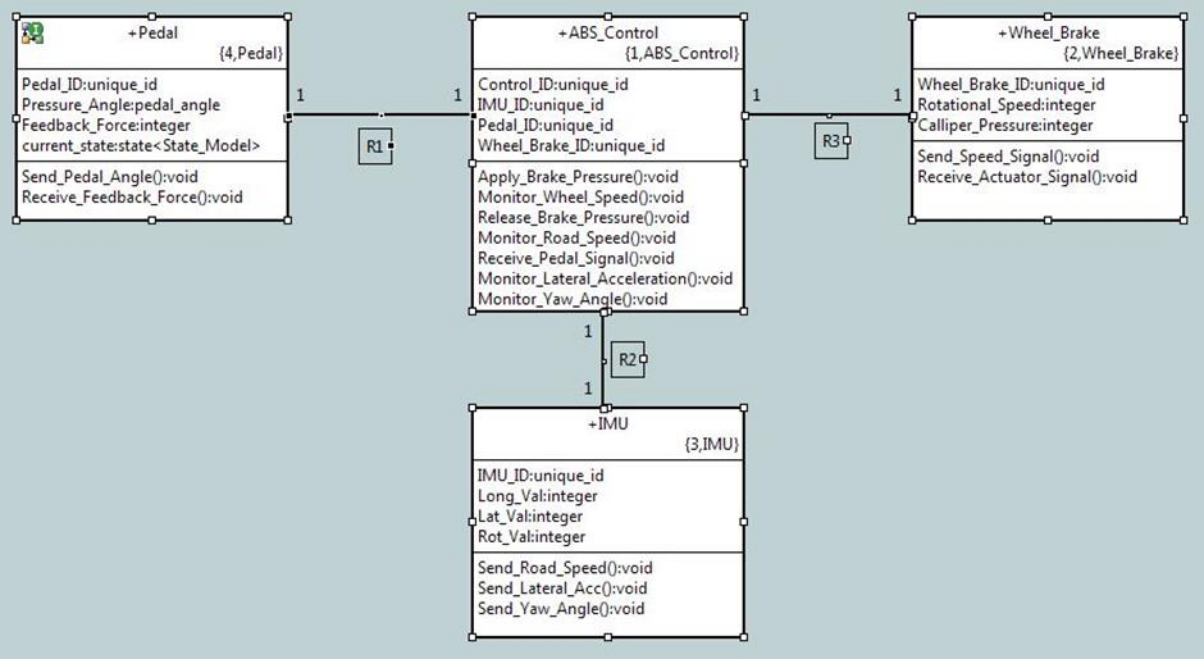


Figure 4-3: xtUML package diagram showing classes for the components of the ABS system

Using BridgePoint's xtUML (an open source freeware UML tool), a primitive executable model was built that simulates some of the features and states of ABS and 'C' source code was generated. Whilst this executable cannot generate anything that could usefully be flashed to a μC board, the action language statements and the diagrams can be used to generate stubs of skeleton C code that can be expanded upon manually by the programmer. A state machine is designed to execute the model with specified scenarios. Code automatically generated by this method, for this research, is included in Appendix I (for the state machines) and Appendix N (for the class descriptions).

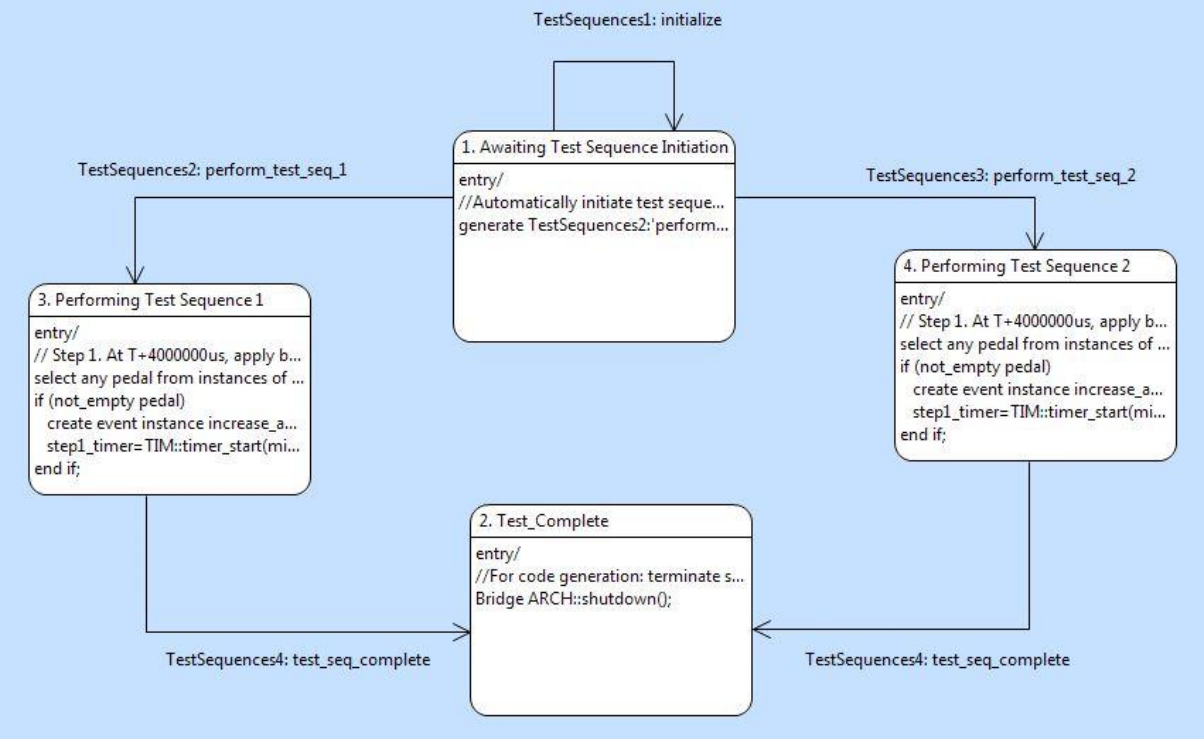


Figure 4-4 : xtUML state machine for a test sequence to run an executable model of the ABS system

The function of the brake pedal is described as two states in a state machine in figure 4-5, increasing and decreasing in pressure.

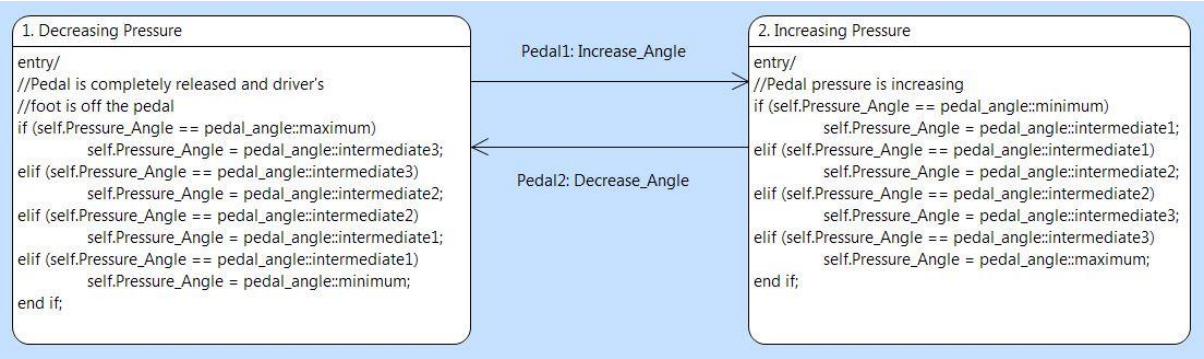


Figure 4-5: Two states of the pedal instance described by a state machine

When an ABS model was attempted in this research with a complete vehicle simulation using Simulink, a dynamical physics model was separated from the controller model as in Figure 4-6. This model was used with a physics model designed outside of the scope of this research using a modified Dugoff tyre model by Ding and Taheri [66] but was limited in its success because of incompatibilities and integration problems due to the nature of the ABS system requiring input and output values at each time step during the execution. This was more of a failure of the human communications than of the ABS model or the physics model.

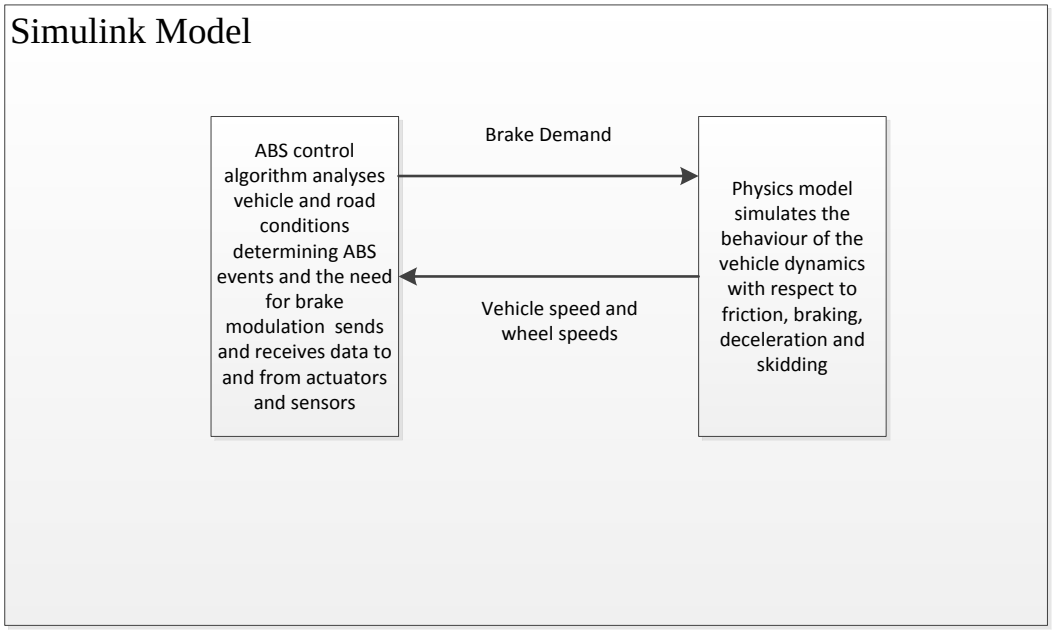


Figure 4-6 : Simulink ABS control model and Vehicle Dynamics Physics model

The controller is programmed in ‘C’ using the S-function builder routine and ‘C’ source code, whilst the physical model is created in Simulink diagrammatic blocks. The integration of the two relies on the input and output of both conforming to an iterative loop scheme whereby at each alternate, discrete, time step a value for the brake demand is calculated and passed to the physical model before the physical model then calculates the resulting changes in wheel speeds and road speed appropriate for the brake pressure demanded and the simulated road/tyre conditions.

In a purely ‘C’ code version of the simulation, the physics model is coded according to rules and equations of constant acceleration with respect to displacement (S), initial velocity (U), final velocity (V), acceleration (A) and time (T) making the presumption that the deceleration of the vehicle can be accurately represented by ‘SUVAT’ equations.

4.2.1 Coding the ABS model

After experimenting with and evaluating UML, Simulink, LabView, Matlab and object oriented paradigms such as Java and ‘C++’, a single codebase strategy was adopted. ‘C’ was chosen because of familiarity with the language, the number of readily available open source compilers and because of its traditional use in embedded software applications on μCs.

Target experimental/evaluation μC boards such as STM 32 F3 Discovery and Arduino UNO are either supported by standard ‘C’ compilers or use a proprietary language which supports standard ‘C’ commands and keywords. Simulink allows the creation of blocks of code (‘S-functions’) that use the ‘C’ language and are translated into wrappers and other ‘C’ source code that can be inlined with Simulink models. Writing for these platforms in a single language reduces the amount of duplication of effort in manually editing source files needed to create code that is common to them all or that can be conditionally compiled for each specific target.

UML was used to automatically generate stubs of ‘C’ source code with some function declarations and class structures obtained from the UML class diagrams and action language statements. The ‘Define_ABS’ function is coded automatically from programmer defined pseudo code and action language typed into the function activity window of the xtUML IDE. Whilst it was possible to generate code in this way, too much effort was required in setting up the UML diagrams and action language for relatively little return that would usefully replace the efforts of a ‘C’ programmer/developer starting from scratch with a diagram drawn in Visio or other such drawing tool.

4.2.2 Cross-Platform software

The ABS model with separate code modules for various features and ECUs is a cross platform executable that can be run as a simulation of a vehicle on separate μCs (for example Arduinos) or as a simulation of the ABS system on a desktop PC.

When running on a PC, each separate feature and the ECU that supports it is modelled in a separate source file. An issue that arises with this model is that the communication bus also has to be simulated, unlike the HILs model with μC boards where a CAN bus is not simulated but actually implemented on MCP2515 chips.

Figure 4-7 shows the virtual and physical network with the communications that are made as point to point messages via the CAN bus. The solid lines and arrows describe communications that necessarily travel via the CAN bus and the dotted lines show the effective direct point to point path of the messages between nodes.

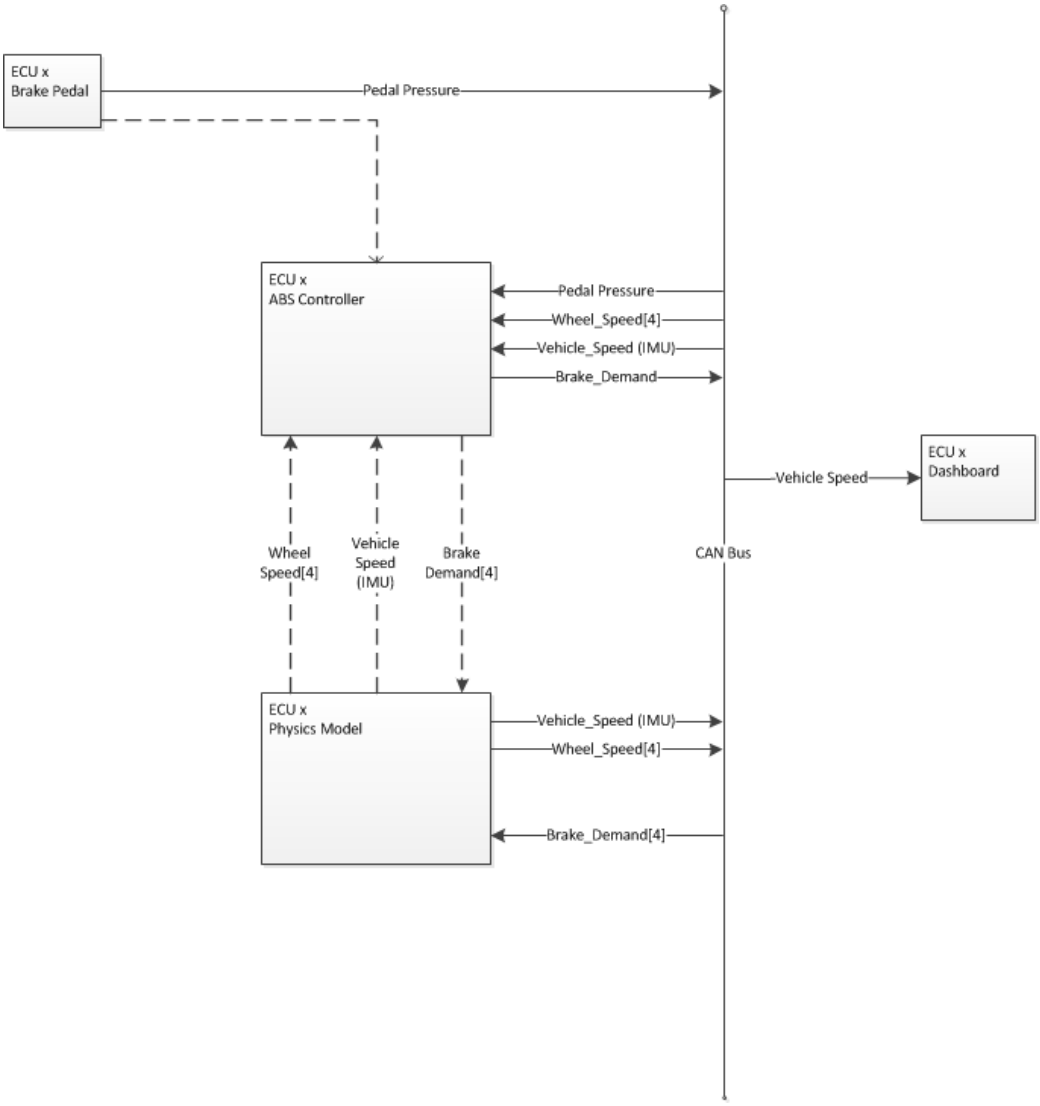


Figure 4-7 : Desktop Simulated CAN model

4.2.3 Sequence of processing in the ABS network

The ABS controller node/program (these could end up on the same node as other programs) polls for brake pedal information. Requests are sent to the pedal, by the ABS controller, asking for the latest pedal position. If the pedal is not being pressed by the driver, the calliper demand is set to zero and o braking will take place.

If a positive pedal pressure is returned to the ABS controller, it will transmit an initial calliper demand value to the calliper node followed by a request for the physics model to interrogate the IMU, the wheels and the recently updated callipers. Whilst there is something slightly artificial about this, because the physics model needs to be executed on a CAN enabled node, it does give the physics model the appearance of acting as though it is passively acknowledging the respective speeds and demands of the IMU, wheels and callipers.

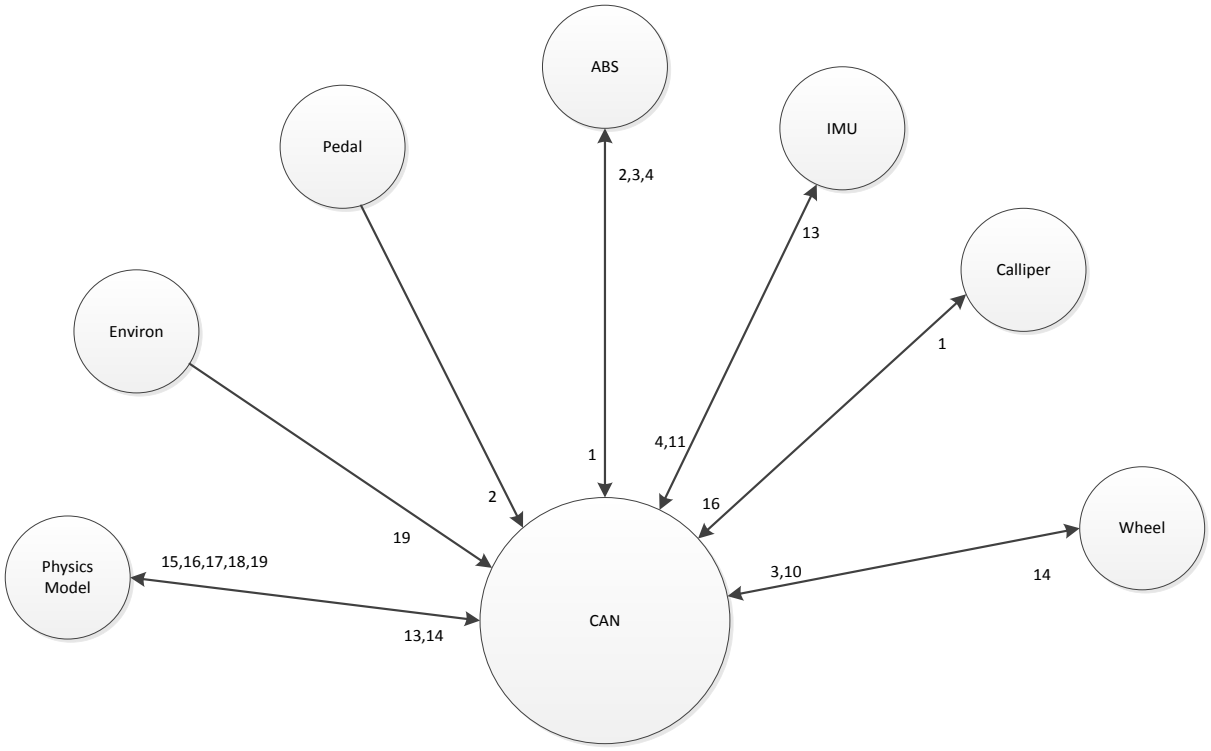


Figure 4-8 : Priorities of CAN messages sent to and from each node in an ABS system simulation

IMU and wheel are shared by both ABS and AMT when integrated. The priorities are calculated as they would need to be in an integrated system before being deconstructed into element parts of both the ABS and AMT systems, where they retain their assigned priority values. This allows each system to run independently and in isolation with no issues when the two systems are integrated. Features such as the communication between the ABS controller and the callipers always retain the highest priority in the system, whether as part of an integrated ABS/AMT or not.

Maximum packet rates for the two systems and the maximum bandwidth are calculated as though each feature sent its message in a round robin fashion. This can be implemented with a time triggered CAN model so as to effectively schedule each event rather than allowing them access to the CAN bus asynchronously.

Pont [2] argues, "In a time-triggered embedded application, the designer is able to ensure that only single events must be handled at a time, in a carefully controlled sequence" and that this has implications for safety because there is no simultaneous occurrence of more than one event that would otherwise increase the system complexity and reduce the ability to predict event-triggered systems' behaviours.

In this research, synchronisation of the nodes of the ECU networks is managed by continuously adjusting the bit sample point during each bit time and continuously resynchronizing the internal time base with the received bit stream as in Voss [15]. In Pont's

Time Triggered CAN example [2], a scheduler shares a single clock between various processor boards with one being the master (with the accurate clock) and all others being slaves that receive clock tick messages from the master. The time triggered system can accurately predict the worst case of message timings and deadlines, useful in safety critical systems with hard deadlines. Figure 4-9 shows the direction of flow of data between nodes of the ABS system with the requests shown as dotted arrows emanating from the node that makes the request and the solid arrows showing the direction of data flow in response to the request or as a direct result of a calculation or update that this node needs to transmit to inform other nodes in the network.

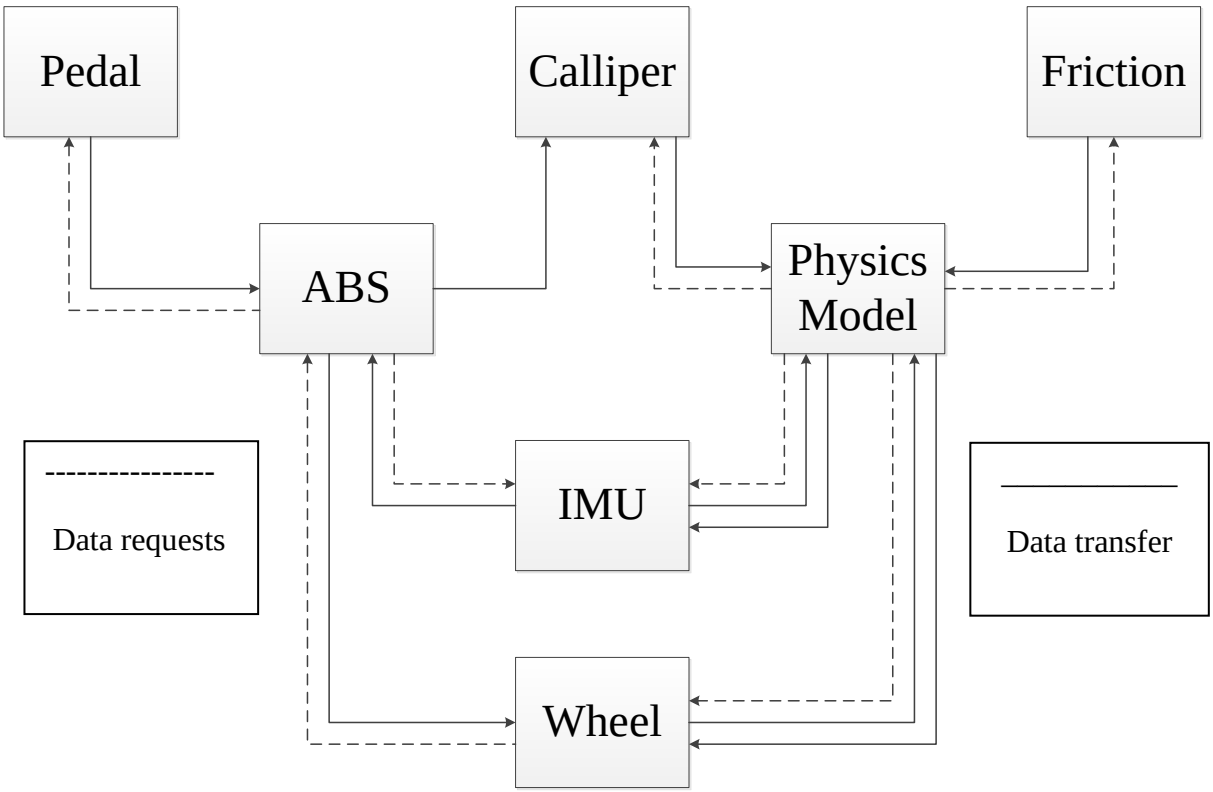


Figure 4-9: showing the flow of data and the messages that request data from other nodes

The wheel speeds and road speed are requested from the IMU and wheels nodes/programs. When the calliper demand, the wheel speeds and the road speed have all been successfully transmitted to the physics model, it calculates one cycle over a predetermined time slice and applies the mathematics/physics of friction and deceleration due to the brake demand and the current speeds (wheels and IMU).

Newly calculated speeds are transmitted to the IMU and wheels so that those nodes now store more recent values than before the application of the physics model function. The ABS polls again to determine whether the brake pedal is being pressed and, if it is, the ABS will use the new road speed and wheel speeds to determine whether or not the current brake pedal pressure exceeds the appropriate brake calliper demand and, if so, modulate it before calling

the physics model once again. When the brake pedal is no longer being pressed, the ABS will receive a zero value from the pedal node and no ABS processing will take place, after which constant vehicle speed will be assumed.

For example, the ABS controller node (ABS) interrogates the brake pedal node (Pedal) and the current value of the brake position/pressure is returned directly to the ABS node. Usually, an interrupt would provide data to the ABS controller via an event driven paradigm but the sending and receiving of CAN messages means that this needs to be managed by continually polling for brake pedal values that are greater than zero and/or different from the previous value. Constant brake pedal values greater than zero cause the ABS function to repeat, until either the pedal pressure changes, the vehicle stops, or the pedal is released completely.

One notable difference is the relation between the ABS and the callipers which is a one-way only communication of data calculated by the ABS and transmitted to the calliper node as a demand for the callipers to be applied or released. In fact, the only real two way transmission of data between nodes is between the physics model and the IMU/wheel-speeds nodes. Two way communication is present between all nodes other than the callipers (which receive actuator signals only) so that, hierarchically, nodes and functions that require data from lower nodes can request them rather than passively waiting to receive a message triggered by an event. This shift from event triggered CAN to a polled CAN reduces the traffic on the bus but is traded of against the time taken per message. Provided the messages are sent quickly enough to meet deadlines, this is not a problem.

In the ‘Desktop Complete ABS CAN Simulation’, messages that are sent to the CAN bus must be passed from the code representing any individual feature/ECU and placed into an area of memory that simulates the CAN bus where it can be accessed by any pieces of code representing other ECUs. This means that there is a global storage area declared in the desktop code that cannot be implemented in the same way in either the HILs or final operational model.

4.2.4 Separation of Concerns

The principle of ‘Separation of Concerns’ [67] deals with separating a computer program into distinct sections, so that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. A concern can be as general as the details of the hardware the code is being optimized for, or as specific as the name of a class to instantiate.

By separating the concern of ‘control’ from the concern of ‘behaviour’, it was possible to write code that would be contained in a single monolithic source file or modularised to be compiled and linked into a single executable program on the desktop PC for verification purposes but which separated the functions for control from the functions for behaviour. Further separation was then possible into programs on individual μC boards with one board acting as the controller and another simulating a vehicle’s dynamical behaviour, using the same lines of source code as before. This is mapped to the distributed ECUs as self-contained

code segments that run in isolation and are called from a ‘main’ function. Any number of these code segments can then be placed either alone on its own ECU or with other code segments that are also called from the main function. For example, in pseudocode, a sequence of functions (for example) could be written into a single source file and a single main function, thus,

```
main_function(){
        send_brake_pedal_value;
        calculate_brake demand;
        send demand to callipers;
        calculate_wheel_speeds;
        calculate_road_speed;
}
```

These could then be separated into individual functions to simulate individual ECU functionality that reside on one ECU or in a single source file on a desktop simulation, thus, still in pseudocode,

```
brake_pedal (){
        interpret_pedal_input;
        send_pedal_value;
}

ABS_control(){
        receive_pedal_value;
        calculate_brake_demand;
        send_demand_to_callipers;
}

physics_model(){
        receive_brake_demand;
        calculate_wheel_speeds;
        calculate_road_speed;
}
```

These functions can now be called in turn by a main function on a desktop simulation in one source file or they can each reside in separate source files that are compiled and linked into a single executable program that operates, thus,

```
main(){
        brake_pedal ();
        ABS_control();
        physics_model();
}
```

Whilst this is now an example of modular code, it is still being executed on one processor, either on a desktop PC or on a single ECU. The final separation of these is when they are

compiled as separate executables on their own ECUs. They run independently of each other but have the ability to communicate with an intermediary controller or a communications bus and messages can be sent from any node to any other node either directly or via some fourth medium.

4.2.5 Simulation of ABS and vehicle physics

By coding an ABS controller model in 'C++' it was possible to create a physical model in a separate executable 'C++' program that receives input from the braking demand (amount of calliper pressure applied to the brake discs) and outputs vehicle-speed and wheel-speed back to the controller.

In this way, a software simulation of a vehicle was created so that the electronic vehicle control systems could be developed to a very advanced stage while better and more realistic physical models could be developed in isolation before being integrated with the controller. Even if the physical model simulator did not return realistic values to the ABS controller (based on the given input to the physical model and values that would be expected) it is still a good test of the ABS controller model, in that it provides known values for input to the ABS controller whose output values can be verified, whether or not they are correctly interpreted and updated by the physics model. The physics model becomes a test tool for the ABS controller and the physics model can be developed in an agile way.

Code was developed as a command-line application, in Visual Studio 2010 on a Windows PC and then recompiled and run on a Linux-based HPC service, using the g++ compiler.

To verify the ABS controller software written in 'C/C++', the physical model can be created either in 'C/C++', to make integration simpler, or in another language/paradigm such as Matlab/Simulink to make the programming easier for an engineer with expertise in these languages but still with the ability to generate 'C/C++' code to be integrated later.

This is at the heart of the problem of 'stove-piped' engineering/development, which has in itself led to ever growing numbers of ECUs in vehicles. Features being realised on individual ECUs are integrated at a late stage of vehicle manufacture. Engineers who have domain expertise will prefer (or indeed be asked) to develop systems in isolation with less regard for integration at an early stage, allowing them to work with tools and in ways with which they are familiar. The task of integrating these many features (upwards of 100 in some cases)

With a tool such as Matlab Coder, 'C/C++' source code can be auto-generated and a software engineer can decide whether to turn this into libraries, executable code, separate source files within a project, or to simply cut and paste the source code into existing 'C/C++' source files for compilation. This cutting and pasting could be automated to create a translation from one source file to another with both manual and automated processes to make 'porting' the code successful across different platforms. Matlab Coder and Matlab Compiler can be used to

generate executables, libraries and stand-alone ANSI ‘C’ code that can compile directly on to target µC boards.

4.2.6 Issues with portability across hardware/software platforms

As an example of the difficulties of attempting to create the same fully operational executable software on more than one platform, a simple program was coded for a windows based console application on a desktop PC and was converted to a version of the code that would perform the same functions on a target controller board (Arduino).

Although the language of both platforms is ‘C’/’C++’, there are commands for printing to a windows console in ‘C’ (‘printf();’) that do not work on the serial monitor of Arduino’s development and debugging environment. The ‘Serial.print()’ command replaces ‘printf()’ but lacks some of the formatting capabilities. To translate from one to the other requires a lot of manual intervention or an algorithm/script or compiler directives with both lines of code existing in the same source files that can make the necessary changes from one platform to the other, automatically. For example, having declared the wheel speeds as

```
double wheel_speed[4]={29.745, 28.933, 29.074, 28.247};
```

then to print out the value of the four wheel speeds to two decimal places and separated by a leading space on a Windows console application, the commands would be (for example)

```
printf("\nThe four wheel speeds are : ");
for(int k=0;k<4;k++)
{
        printf(" %.2f ", wheel_speed[k]);
}
```

and to perform the same task on the Arduino serial monitor would be

```
Serial.print("\nThe four wheel speeds are : ");
for(int k=0;k<4;k++)
{
        Serial.print (" ");
        Serial.print (wheel_speed[k],2);
}
```

In a cross-platform source file, code that would print the correctly formatted output on whichever of the two systems it was being compiled for would (for example) use the compiler directive “#ifdef” or “#ifndef” (if defined, or if not defined) to inform the compiler of the target platform, that is, the code would look like

```
#ifdef DESKTOP
printf("\nThe four wheel speeds are : ");
for(int k=0;k<4;k++)
{
        printf(" %.2f ", wheel_speed[k]);
}
#else
Serial.print("\nThe four wheel speeds are : ");
for(int k=0;k<4;k++)
{
        Serial.print(" ");
        Serial.print(wheel_speed[k],2);
}
#endif
```

When the code is compiled, an executable that will only ever perform one or other of the desktop or Arduino code segments will be created. Thus, it is possible to create two compilation ‘makefiles’ one with the desktop definition and one without so that two executables exist each of which is specific to its own platform.

4.3 The ABS Controller

The ABS controller is a software/hardware device that controls the braking demand in a vehicle to prevent the wheels from slipping or locking in the event of low grip or harsh braking. A software algorithm uses real-time vehicle-speed and wheel-speed data during braking events to monitor and modulate braking demand for safe, controlled, braking when the driver has braked too fiercely for the road/tyre conditions.

4.3.1 Principle of operation of ABS controller

The purpose of the ABS controller is to take input from HMIs or human computer interfaces (HCIs) such as a brake pedal (which may itself have an embedded μC) and to calculate a brake calliper pressure demand value and pass it to the brake calliper actuator. This can be transmitted to the actuator software via one of the candidate communications protocols (for example CAN) and can be actuated at the callipers/discs by solenoid, motor or hydraulic pump.

4.3.2 ABS controller algorithm

The algorithm for the ABS controller, in figure 4-10, compares individual wheel speeds with other individual wheel speeds, with an average of all wheels speeds and with the vehicle speed obtained from an inertial measurement unit (IMU). If there is a significant difference (referenced by an arbitrary tolerance metric) between the average wheel speed and the vehicle speed, the ABS controller will trigger an ABS event that will reduce the pressure of the brake callipers on the brake discs – either directly by activating the brake callipers electrically (in a 'brake-by-wire' system) or by reducing pressure within the brake lines in the case of a hydraulic system. Further, iterative, measurements of the wheel speeds and vehicle speed will allow the ABS controller to modulate the calliper pressure demand until all wheels are fully rotating before applying brake pressure again – for as long as the brake pedal is depressed or until the vehicle comes to a halt.

For example, in the simplest case, from an initial vehicle velocity of, say, 30m/s and with all wheels rotating at a similar speed, assume a brake demand is applied at the brake pedal equivalent to a braking force of (for example) -10m/s/s on the wheels and that this is too great for the friction force between the tyres and the road. An ABS event is triggered that causes the wheels to lock and the wheel speeds are now out of alignment with the road speed measured at the IMU. The ABS controller compares these values and the difference in speeds, 30m/s from the IMU and zero from all four wheels, initiating an intervention between the pedal and the callipers and the release of the brake callipers from all four brake discs. This allows the wheels to begin rotating again due to friction alone (although, later, the engine and/or motors could be used to reduce the time it takes for the wheels speeds to match the road speed) and reach the same speed as the vehicle – which will have slowed a little due to the skidding that has been allowed to happen in the moments that the wheels were locked. The ABS controller applies the brakes again to the level demanded by the driver at the pedal and, if necessary, initiates another ABS intervention until either the brake pedal pressure is reduced or the vehicle has stopped. In an asymmetric ABS event, where not all wheels have either locked or under rotated, the ABS controller compares the wheels speeds individually against the average wheel speed and the IMU speed.

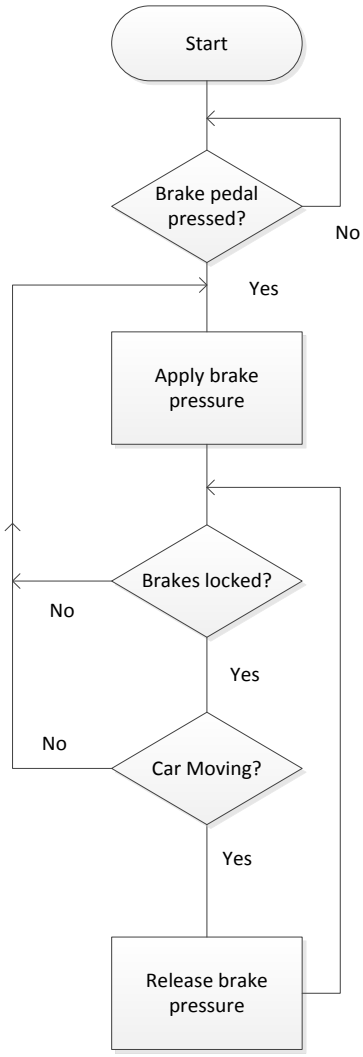


Figure 4-10 : Flowchart of operation of ABS algorithm

4.4 Physics Model of vehicle behaviour under braking

Braking events in a road vehicle can be divided into five main categories.

Normal braking, maintaining control over wheel speeds and vehicle speed to a controlled full stop or to a reduced road speed for continued safe driving.

Slip – the under rotation of one or more wheels causing slipping that can be recognised by the vehicle electronics and corrected by differential braking at one or more wheels without skidding.

Asymmetric Locking – the locking of one or more but not all wheels requiring the intervention of ABS to modulate the brake pressure at the wheels/callipers to maintain stability and control over steering and to slow the car or bring it to a complete halt.

Skidding – the locking of all wheels to the point where the vehicle is no longer in control and is slowing only due to friction between four non-rotating wheels, effectively turning the vehicle into a skid block.

Rolling – the recovery from an ABS event where the brake demand at one or more wheels is reduced or the calliper is completely released, allowing the wheel(s) to rotate up to the vehicle's road speed due to the friction from contact between the tyre and the road surface.

The purpose of a physics model as part of the ABS model and simulation is to produce the feedback of the physical behaviour of a road vehicle that would verify the ABS controller model without the need to implement the controller in a real vehicle. The physics model takes calliper-demand, road-friction and initial wheel/road speeds as inputs and calculates the physical effects of these over time (fractions of a second) to simulate the reductions in speed of both the vehicle and the wheels. These are fed back to the ABS controller as they would be from speed sensors on a real vehicle and the controller can act exactly as it would in a software simulation or HILs or when fully implemented in a vehicle.

Kudarauskas' research into the braking behaviour and deceleration values of a variety of road vehicles [68] records the braking deceleration to be as high as 9.5m/s/s with ABS, around 7.5m/s/s without it and 8.5m/s/s in a general case. These figures have been adopted by this research as midpoint values and some variation has been allowed in the simulation of vehicles in this research.

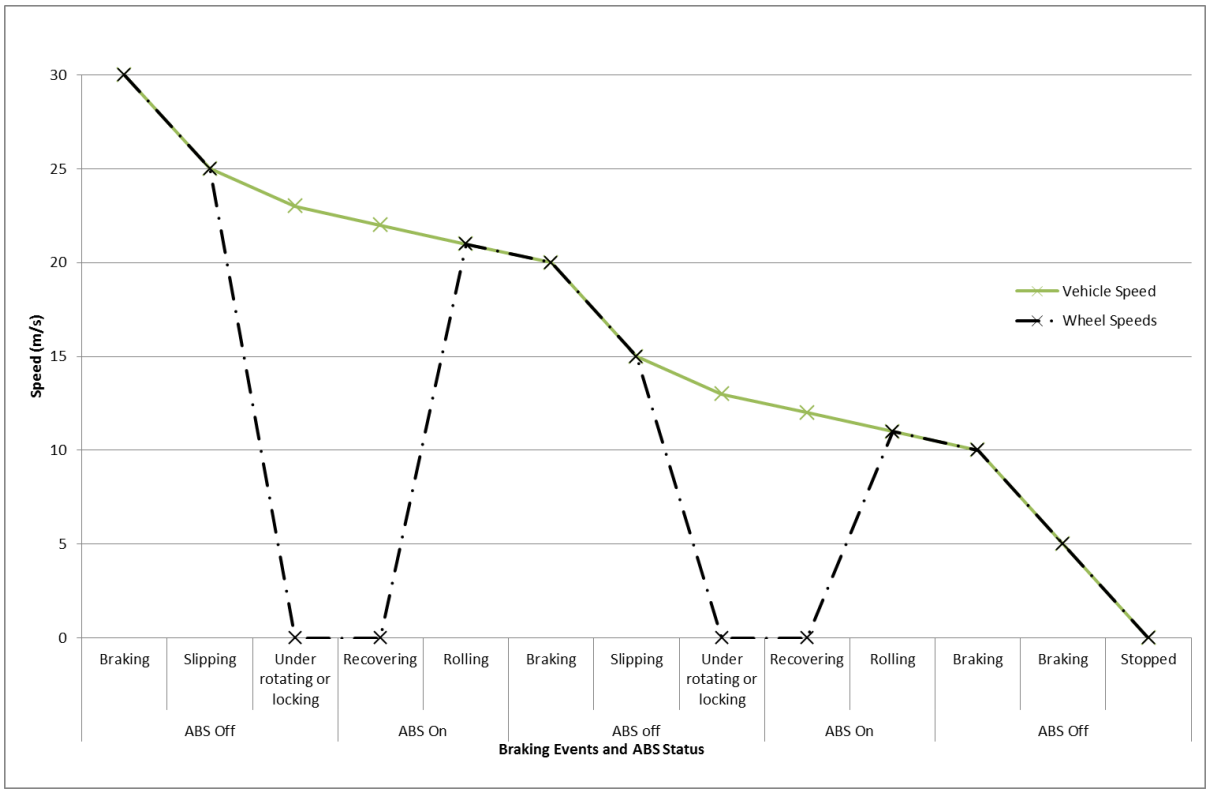


Figure 4-11 : ABS braking event with all four wheels behaving in a similar symmetrical synchronised fashion

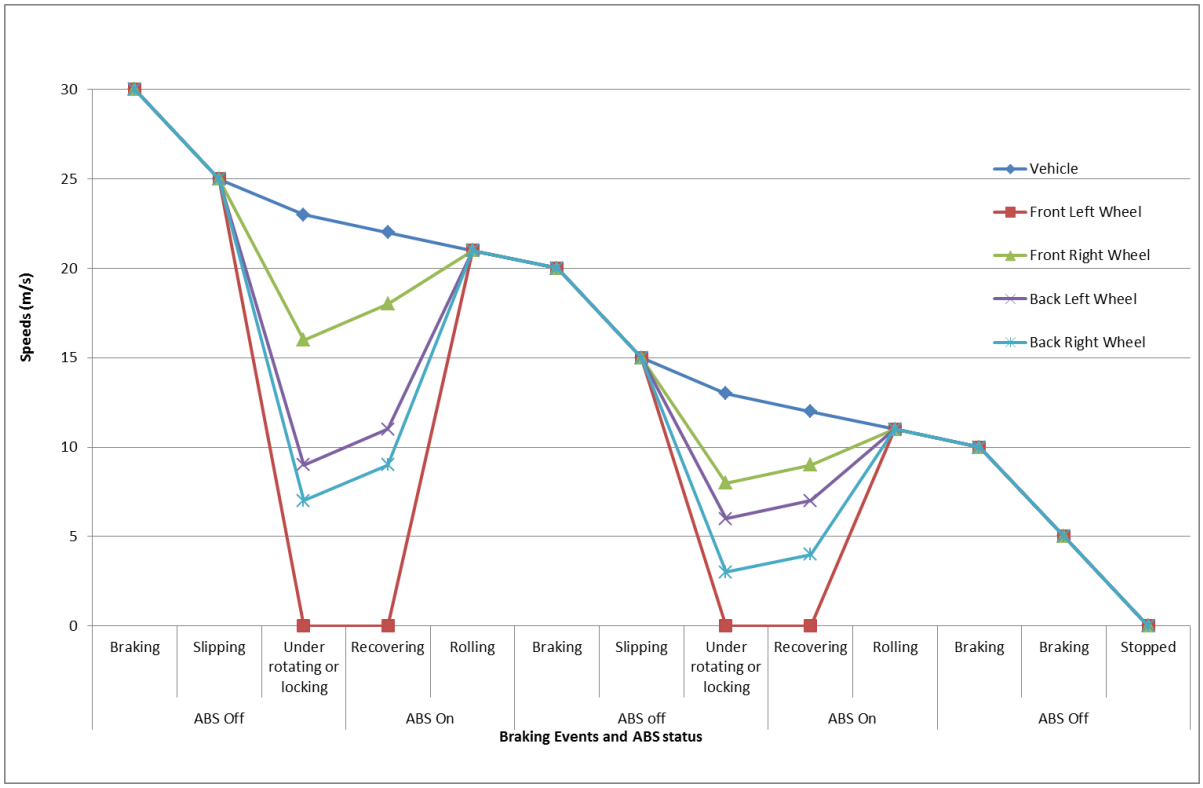


Figure 4-12 : ABS braking event with wheels behaving independently and asymmetrically

4.4.1 Principal of operation of physics model

To reduce costs by verifying the ABS controller in software, the physics model takes the place of a physical road vehicle and its features that would send or receive signals and messages to or from ABS, such as brake callipers and speed sensors. It provides values for the wheel speeds and road speed of the vehicle, which would normally come from physical wheel speed sensors and an IMU. These values are visible to the ABS controller via simulated communication protocols in desktop computer applications or via actual CAN messages between target μC boards connected to a shared communications bus.

Using SUVAT equations of constant acceleration, it was possible to model vehicle behaviour under braking events without going down to the level of detail of tyre models proposed by Ding [66] and Pacejka [69]. An assumption is made that, whatever the effects of tyre deformations and tyre structures or the coefficients of friction between the tyres and the road surfaces, the physical model serves only to produce values to be analysed by the ABS controller, that is that the wheels are rotating slower than the forward motion of the vehicle or that there is a mismatch between the speeds of different wheels on the vehicle during a braking event.

Whatever the causes of these mismatches, under rotations, locking, skidding that are generated by the physical model, the function of the ABS controller is to modulate the braking demand by reducing or increasing the brake calliper pressure on the brake disc. Provided that this causes a safer braking event with locking kept to a minimum, the model is sufficient to validate the ABS controller, the amount of executable machine code needed to

run it, the memory capacity on the processors and the network topologies that optimise the communications between the controllers and the system hardware.

For the ABS event of wheel-locking, an aircraft model presented by Wang [70] is used to represent a wheel rotating up to the vehicle speed from zero. This is the equivalent of a non-rotating wheel on an aircraft's undercarriage upon landing as it touches the runway and accelerates to match the speed of the aircraft. By assuming similar levels of grip/friction on the car as to those between the aircraft's wheel and the runway, an approximate value for the acceleration of a locked road vehicle's wheel can be estimated.

Assuming similar coefficient of friction between aircraft tyre and runway as for that between car tyre and road, the acceleration of the car's road wheel is approximately 1080 ms^{-2} given that it can accelerate to 75.6 ms^{-1} in 0.07 seconds. In the specific case of the road wheel, it is not expected to accelerate to a speed greater than 40ms^{-1} and acceleration is assumed to be constant while both acceleration and speeds are assumed linear.

Figure 4-13 shows the constant acceleration of a locked wheel from the instant the brakes are released by the ABS (in red) up to the moment the wheel speed and vehicle speed are matched, at which time ABS will continue to analyse the wheel speed while the brake pedal is pressed. Wheel speed is measured in units of linear velocity as an equivalence of angular velocity for a wheel of a given diameter. For example, for a tyre radius of 8 inches (based on standard 16" diameter tyres) the calculations, in SI units give

radius=8*0.254 metres = 0.2032m
circumference = $2 \pi r$ = 1.2767m
velocity = revolutions per second * circumference = $2 \pi r$ * revs = revs * 1.2767
giving an equivalence for a 16" diameter tyre of 1 revolution per second = 1.2767 ms^{-1}

Data for the aircraft wheel, in grey in figure 4-13 are used to model road vehicle wheel speeds up to 40.5ms^{-1} and vehicle wheel speeds accelerating from stationary when the brakes are released are modelled on an interpolation of aircraft wheel speeds upon touchdown when landing.

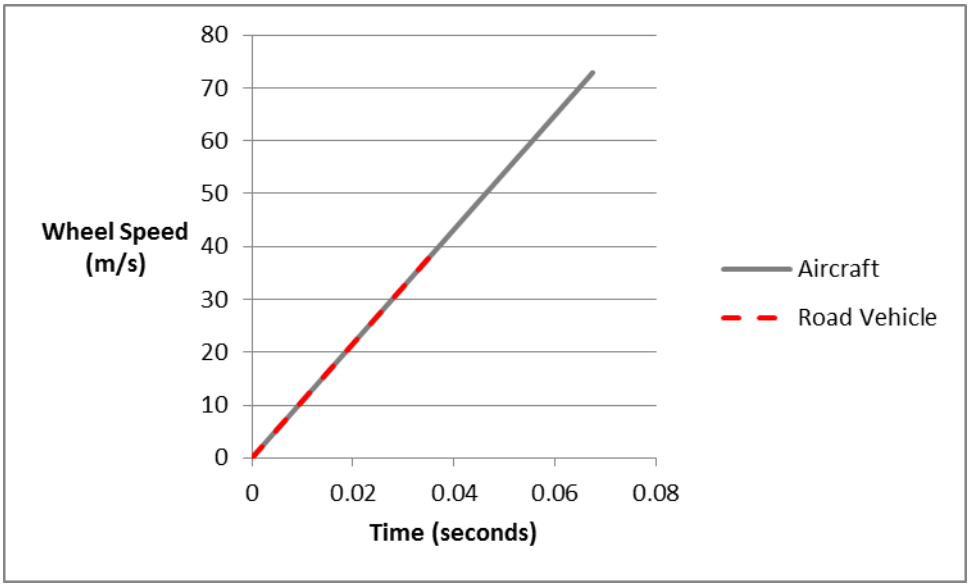


Figure 4-13: Combined aircraft wheel speed and interpolated vehicle wheel speed

4.4.2 Physics model algorithm

The physics model (in a simulation of the ABS) produces values for the vehicle's dynamic behaviour (road speed, wheel speed) based on parameters that are passed to it from the environment model (road surface, road condition, weather, tyre condition) and from the ABS controller (braking demand as a value of the calliper pressure). These, combined with the current road speed and wheel speeds will simulate the reduction in wheel speeds and road speed that would be exerted as a direct result of the pressure applied by the callipers to the discs during a braking event.

Considering two extreme levels of road friction from very slippery (low coefficient of friction between road and tyre) to very 'grippy' (high coefficient) a different algorithm is used in each case with one being specifically for the lowest coefficient (coded as zero) and another for all other values greater than zero.

To keep the model relatively simple to understand, the phases of a braking event are separated into an initial phase, which necessarily begins with 'ABS off' and includes a braking phase where the wheels and vehicle both begin to slow, followed optionally by 'ABS on' and finally and necessarily by a 'cadence' phase which may be a continuation of the initial braking phase if no slipping or locking has occurred and there is no ABS event. Normal driving is considered to be outside of a braking event.

'ABS off' is divided into three phases…

1) Braking phase - braking that slows the vehicle before any loss of grip between tyre and road. A combination of the road/tyre conditions (measured as a coefficient of friction between the tyre and road) and the brake demand (requested by the driver at the brake pedal) produces the value for a deceleration force to be applied to both the wheel rotation and the forward longitudinal velocity of the vehicle. When the brake

pressure is significantly great to overcome the friction between the road and the tyre, there will be a departure of deceleration values for the wheel compared with the vehicle. This is a representation of harsh braking that will result in either slip or wheel-locking and an 'ABS on' event.

2) Slip phase (optional) – braking that has less effect due to an under rotation of the wheel compared with the vehicle speed but which doesn't include any locking or skidding, but before any ABS event has been invoked

3) Locking phase (optional) – braking that has locked one or more wheels and produces a skidding wheel or a sliding vehicle, but before ABS is engaged

'ABS on' phase (optional) – the reduction in or the complete removal of brake calliper pressure to allow one or more wheels to increase in rotational speed. This is defined by two phases…

1) Recovery phase (optional) – where the brake is released completely due to locking and a wheel accelerates from zero up to the vehicle's road speed

2) Rolling phase – ABS is active and the calliper pressure has been released to allow for ABS on phase 1 above. The wheel has good grip with the road surface and is accelerating up to the road speed of the vehicle

The cadence phase occurs when the friction between the road and tyre has stabilised and returned to a rolling resistance state and the brake pressure combined with the road speed and tyre grip produces effective, non-slipping, non-locked, braking. This is effectively the beginning of another 'ABS off' phase and is characterised by good braking which leads to no further slipping or locking during that one braking event. This can be considered an 'ABS off' initial phase that results in either the vehicle stopping or continuing under control without the intervention of the ABS.

The assumption is that for a constant time step between each iteration of ABS control and vehicle behaviour, the time taken for any wheel to begin slipping or to lock completely will vary according to vehicle speed, road condition, brake pressure and that the deceleration experienced by the vehicle and the wheels will initially be matched for a brief portion of the time step before the slipping or locking phase that will last for the remainder of the time step and cause the vehicle and the wheels to undergo different decelerations because of the lack of grip between the wheels and the road. Torsional stiffness of axles, wheel hubs and the wheels, as well as deformations of the tyre wall/tread are ignored with respect to their negligible effect on differences between wheel speeds and road speed or on braking behaviour generally for the purposes of this research.

4.4.3 Detail of physical model and nominally zero friction

The physics model used for the ABS system takes input parameters for individual wheel speeds, friction between the road and each individual tyre, the speed of the vehicle and the

calliper pressure at each individual wheel's brake disc. These 13 parameters are then processed within a specifically simulated time slice to obtain values for the changes in wheel speed and vehicle speed after that time interval. The functions determining the changes of speed on the vehicle and on each of the four wheels use the constant acceleration equations of classical mechanics.

The values for acceleration (negative acceleration in the case of braking) were determined by a combination of the brake calliper pressures and the friction between the road and the wheels. If the calliper force on any individual wheel exceeds the value of the friction between that wheel and the road, the braking force on that wheel is set to an arbitrarily high value of 100 which in most cases will cause the wheel to instantly lock, simulating harsh braking on a slippery surface. Otherwise, the braking force is set to the same value as the calliper pressure multiplied by some deceleration constant (for example 8 – units not specified).

Having set this braking force for each wheel, the wheel speed is reduced by the product of the braking force and the time slice. This performs the operation of the SUVAT equation (4.1) where u is the initial velocity of the wheel, v is the final velocity, t is the simulated elapsed time and a is negative acceleration equal to the braking force. If this reduction were to result in a negative value for the new wheel speed, it would be set to zero simulating a locking event.

Now that a new speed has been calculated for each wheel, the effect on the overall braking of the vehicle is calculated. An average stopping force that can be applied to the vehicle is calculated from the four individual braking forces for each wheel by simply summing the forces and diving by the number of wheels.

If the wheel speeds are all zero and the vehicle speed is less than the value of a single time step, rather than being a wheel-locked ABS event, the vehicle is considered to have come to rest. Similarly, if the product of the average stopping force and the time slice is greater than the current vehicle speed, the vehicle will come to rest after one time step and all speeds will be set to zero.

Otherwise, if the effect of deceleration does not bring the vehicle to rest, the vehicle speed is reduced by the product of the average stopping force and the time slice, again invoking the effects of equation (4.1). If the new vehicle road speed is calculated to be zero, all wheel speeds are set to zero.

No consideration is given to the materials that make up the braking components or the tyres or road surfaces. Friction values are arbitrary but could easily be coded to directly reflect different materials such as steel, ceramics, carbon fibre, for example. The physical behaviour and deformation of tyres under different acceleration events such as braking or steering are not part of the physical model and nor is the torsional stiffness of axels or driveshafts that might affect the overall braking or the acceleration events referred to later in chapter 5. The coefficients of friction for each tyre are assumed to be constant as are those for the brake discs and pads.

The algorithm that deals with simulating conditions such as icy roads is triggered when the value of the friction variable is set to zero. The zero value is nominal only and represents the lowest probable friction coefficient encountered in any real driving conditions (for example oil, ice or compacted snow). The calculations of wheel-speed reduction and vehicle-speed reduction will use non-zero values to avoid any possibility of 'divide-by-zero' errors.

For a given initial velocity, u (considered being in a forward direction only and therefore a speed scalar), the final velocity after one time step (arbitrarily one second) is calculated by the simple SUVAT equation

$$v = u + at \tag{4.1}$$

where *v* is the final velocity, a is the (negative) acceleration due to the combined effects of braking, friction and gravity and t is the time period for the braking event. Rolling friction of internal components of the vehicle is ignored and any value of *v* that goes negative is treated as zero, rather than allowing for the vehicle to be in reverse motion or rolling backwards.

In this way, a single equation can be used in all eventualities where the road friction takes values from a minimum of 1 up to a maximum of 8 to represent the extremes of slipperiness and high grip.

4.4.4 Simulink models for ABS

The physics model that simulates the behaviour of the vehicle during braking phases is provided in Simulink and the modified tyre equations of Dugoff in Ding [66] are used to simulate the deceleration of a vehicle when the brakes are applied in different conditions (dry, rain , snow etc.) and a graph of the time sliced vehicle speed and wheel speeds is produced. In the Simulink model, time slices from between 1 second and 1 microsecond have been simulated. Changes to the braking torque (Nm) from 0.1 to 0.00001 produce different braking characteristics for the same simulated road conditions. Times of between 0.2s and 1.5s are recorded for the wheels to lock after braking force is applied. The vehicle then slows to a complete stop due to sliding friction with wheels completely locked. This takes about 15 seconds.

An ABS controller algorithm is also provided by this research in Simulink to produce updated brake modulation at every time step if the wheels are slipping, about to lock or (on icy or very wet roads) completely locked. The inputs of the ABS controller algorithm are the four individual wheel speeds, the vehicle speed (simulating an inertial measuring unit (IMU)) and the brake pedal force or angle that is being applied by the driver. The outputs are newly calculated brake torque values to be sent back to the physics model in a control loop. If the brake pedal is not pressed or the vehicle is not moving, no ABS event can be triggered and the ABS controller's brake modulation function is not called. In extreme cases of all four wheels locking while the car is still travelling forwards (skidding) the ABS controller completely releases the callipers from the brake discs allowing the wheels to regain grip and

to begin rotating again. Only when a wheel has matched the vehicle's speed will the calliper pressure demand be restored and the brake applied.

Finally, asymmetric braking where one or more wheels under rotate (based on differences in speed between one wheel and the fastest wheel or between one wheel and the average speed of all four wheels) while at least one remains at or close to the vehicle speed, causes the ABS brake modulation function to reduce the calliper pressures and the braking torque on up to three wheels only. This allows the remaining wheel(s) to brake effectively while the brake modulation in the slipping wheels partially releases the brake callipers thereby reducing the braking torque but not releasing the brakes completely. This increases safety compared with a system that completely releases the brakes on all wheels, as the time for the under rotating wheel to come up to the speed of the fully rotating wheel is reduced and the point at which all four wheels' speeds are equal is somewhere between the speed of the fully and under rotating wheels.

Figure 4-14 to Figure 4-16 show the ABS controller and physics models that are integrated in the Simulink model. In Figure 4-14, the ABS controller reduces the calliper pressure on the front right wheel because it is under rotating in comparison to the other wheels and the vehicle road speed. When this output is fed into the physics model (Figure 4-15) the front right wheel slows by only 0.2% compared with the 0.625% of the other three wheels, thereby reducing the slip in the under rotating wheel.

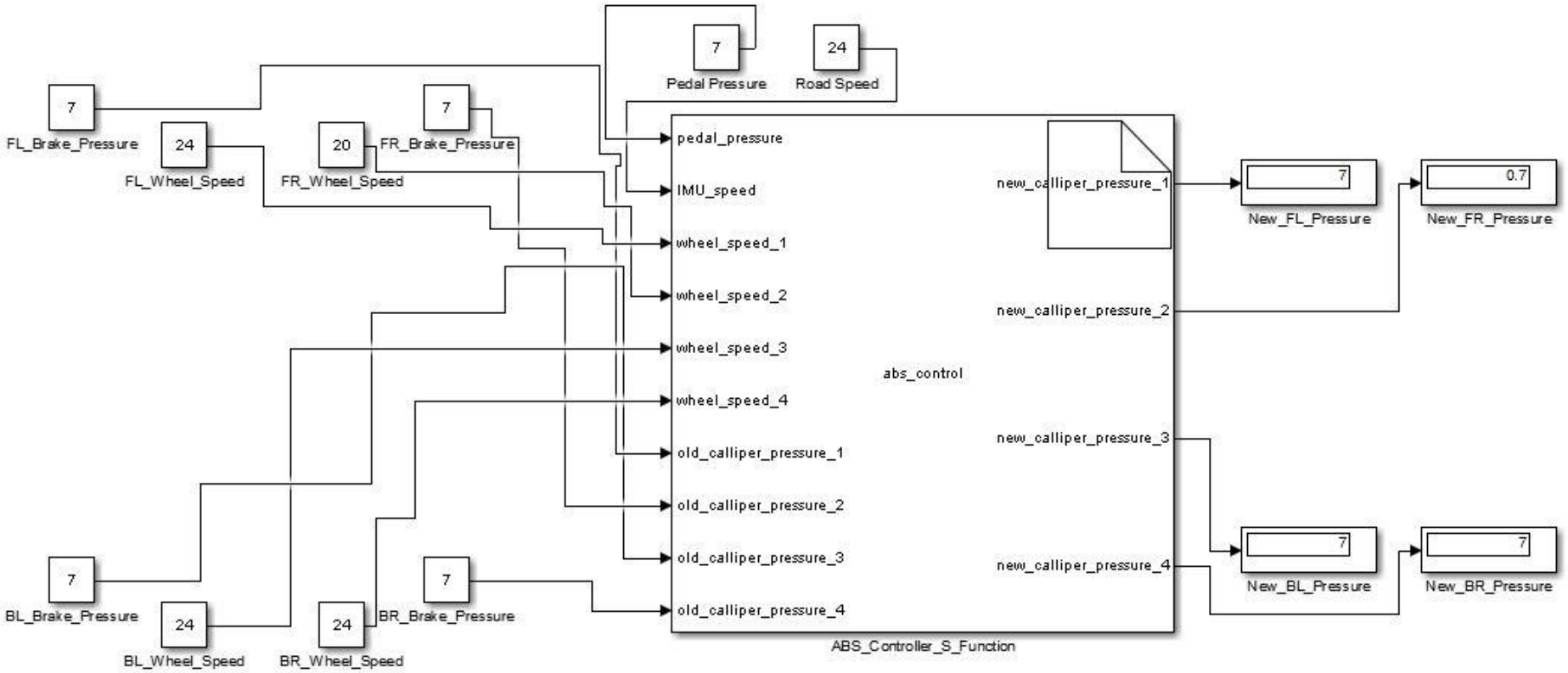


Figure 4-14 Simulink model of ABS controller reducing brake demand on under rotating front right wheel

The scenario in Figure 4-16 is of reduced friction between the front left wheel and the road when the road speed and wheel speeds are matched and the braking demand is equal and uniform across all the wheels. The result is complete locking of the front left wheel and very slight slipping of the other three (23.85 units) compared with the vehicle road speed (23.86 units).

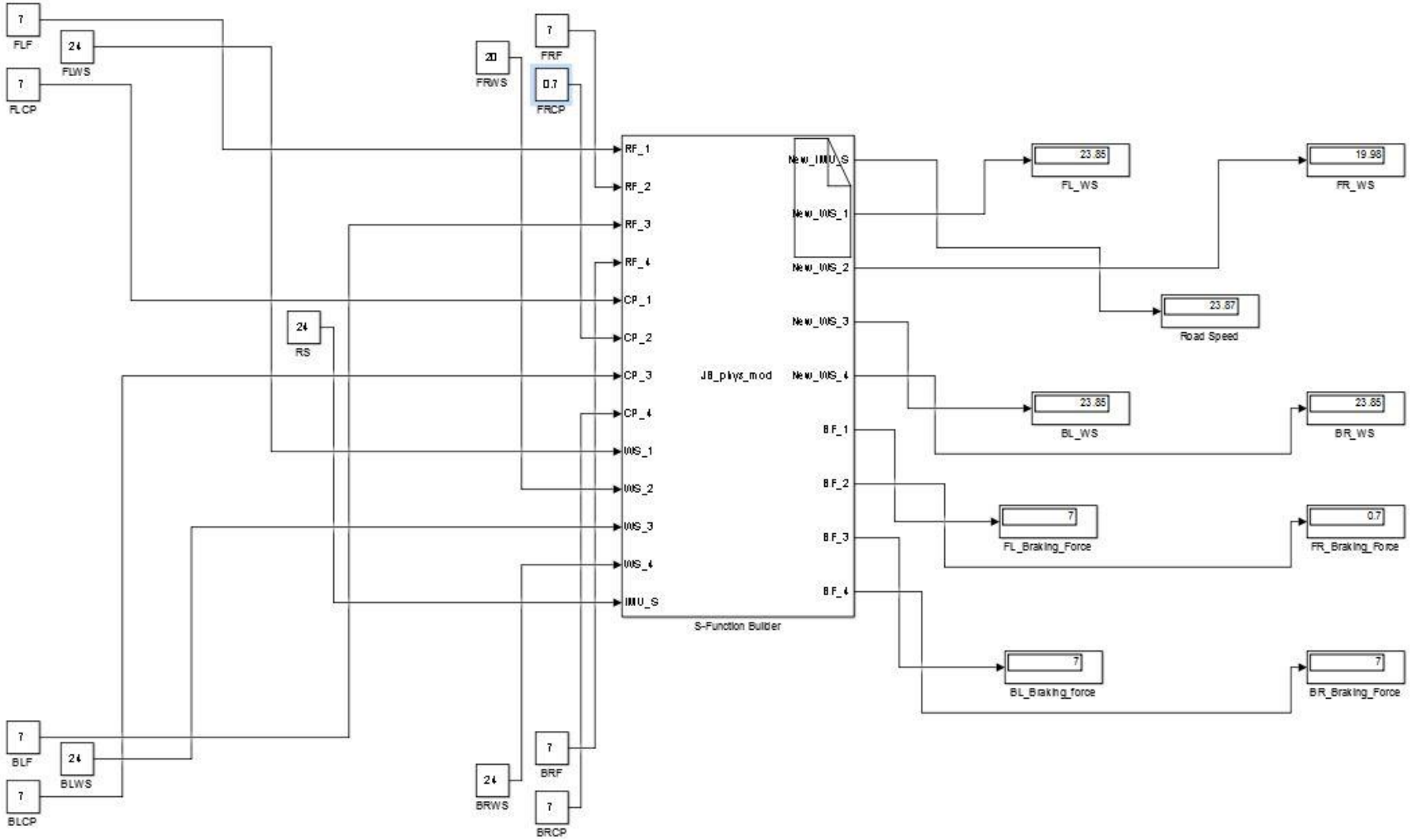


Figure 4-15 : Simulink physics model with reduced speed and calliper pressure at front right wheel leading to slip at all four wheels

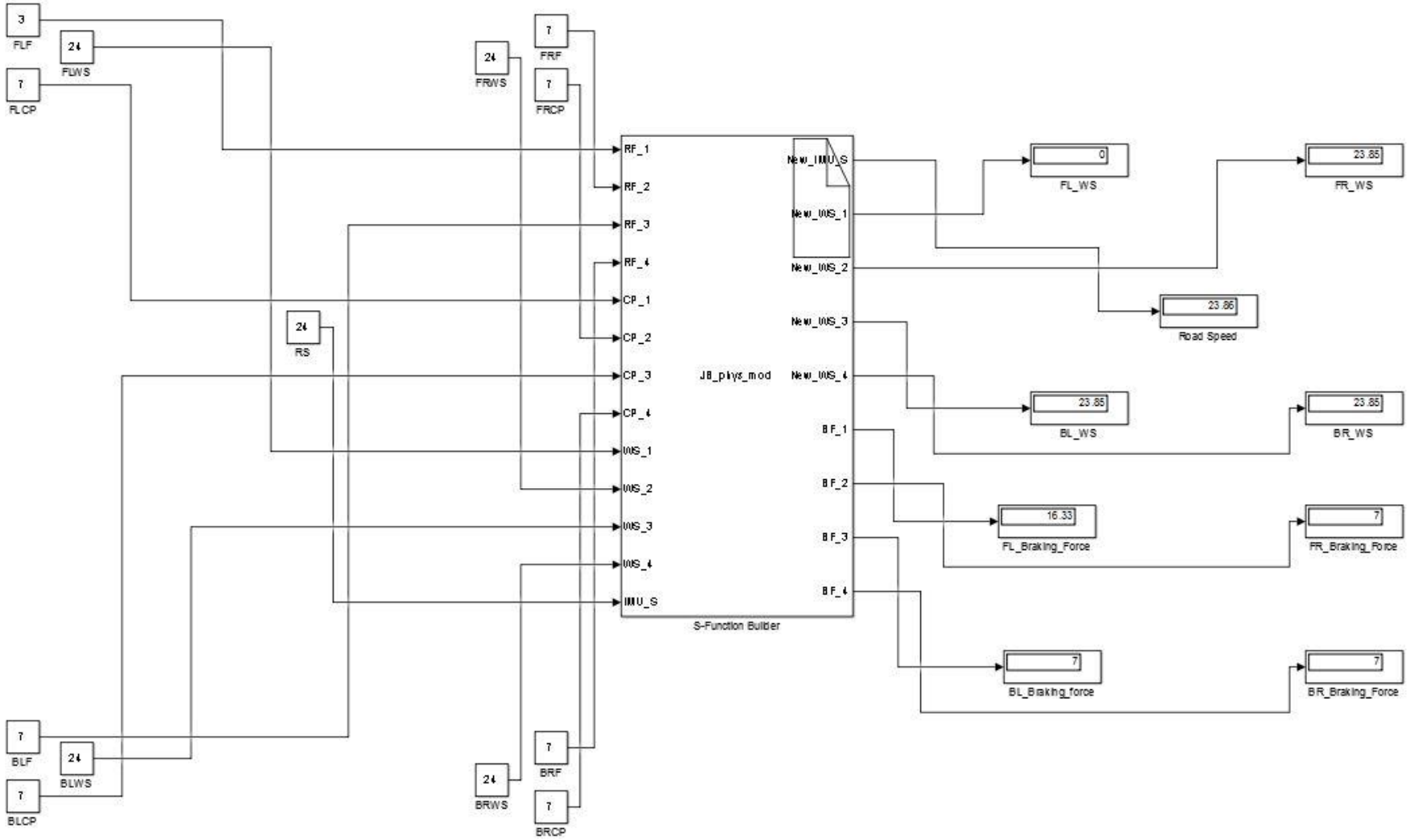


Figure 4-16 : Simulink physics model showing braking event beginning with reduced friction at the front left wheel

In Figure 4-17, wheel speeds for front left (FL), front right (FR), back left (BL) and back right (BR) wheels are shown together with the vehicle speed (IMU) over the last 2.22 seconds of an emergency braking event with asymmetric slip and locking. The friction values for the back left wheel are set up to cause it to behave as though it is either over-inflated or worn. Consequently, it locks comparatively quickly from vehicle speed to zero and is released allowing it to rotate to full speed after approximately 0.6 seconds during which time the back right wheel, also experiencing an ABS event, has pulsed 10 times without locking – due to the coefficient of friction between that wheel and the road being greater than between the road and the front left wheel. The other three wheels share similar friction coefficients and are set up to behave similarly to each other.

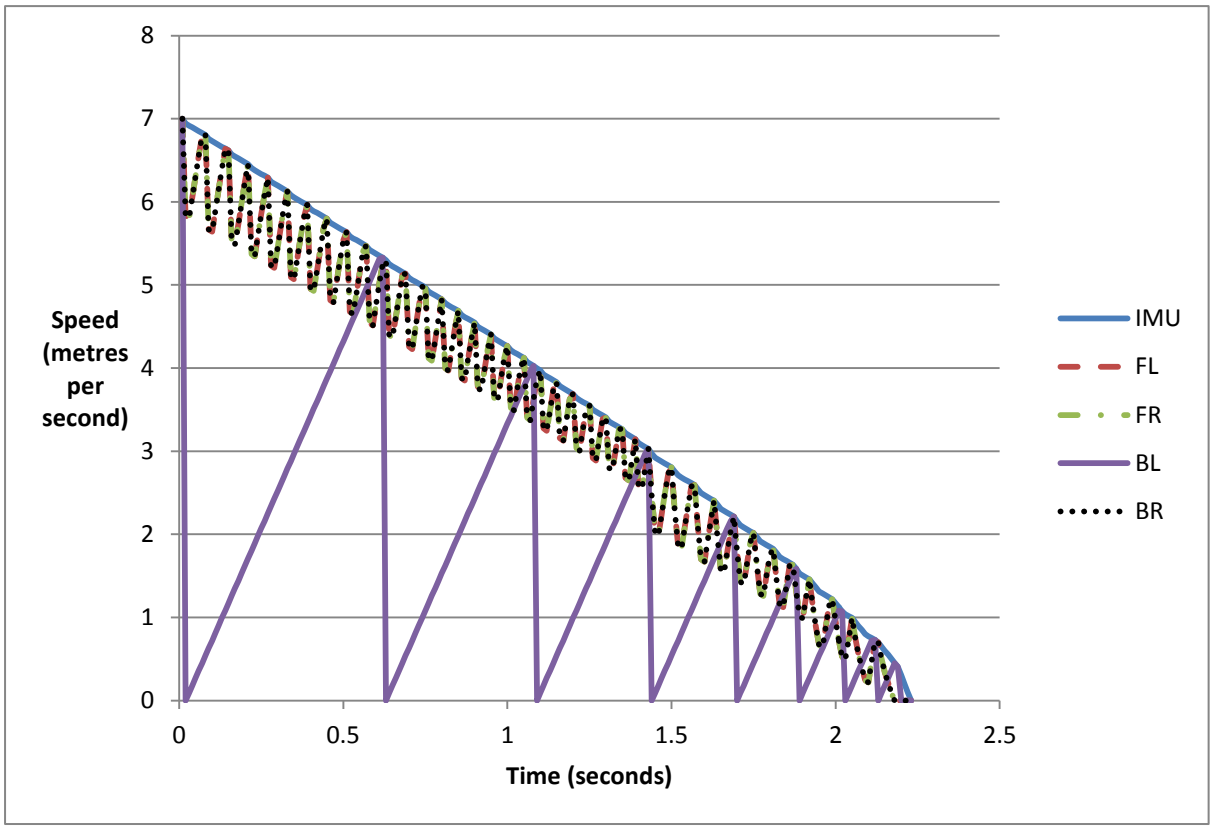


Figure 4-17: Asymmetric braking effect of individual brake pressures obtained from this research's executable ABS control model

4.4.5 Issues associated with simulating separate ECUs in a single hardware platform

Because the behaviours of separate ECUs are being modelled, the programming code needs to reflect the separation of each module, that is a separate module for the ABS controller and for the physical model. However, the temptation to simply write chunks of code that are separated in so much as individual functions and header files but which share common global variables is both great and wrong.

In the real world deployment of the ECUs, there is nowhere for the global variables to reside, since there is no shared memory between the ECUs except for data passed to the communications bus that can be accessed by the other ECUs on the same bus. Any variables in the memory of one ECU are invisible to all other ECUs until they are transmitted on the communication bus which is either physically connected by wire or is remotely connected by some wireless protocol. This is solved by writing code segments that encapsulate the data and which have no scope outside of the functions or procedures to which they belong. In this way, values that would belong, for example, to the brake pedal but should not be seen by the calliper function until they have been passed to and out of the ABS controller would be passed in memory within the parameters of a function call and would then go out of scope when the function returned control to the main program.

4.5 CAN Simulation

Use of a third party, proprietary, CAN simulation package “RTaW-Sim" allows for the simulated execution of real time communications between ECUs and buses with accurate timings. The features supported by the RTaW-Sim Starter edition, as opposed to the much wider range of features in the “Pro” edition, are

Simulation of CAN 2.0A buses
Gateways
Event driven communication transmissions patterns
CAN bus transmission errors
TTCAN

Because of this limitation in the starter edition, the CAN simulation in this research was reduced to measurements of the CAN messages on the Arduino hardware/software simulations for predicting the correct functioning of the hardware implementation.

4.5.1 Maximum message rates and packet rates

Since a standard CAN message is 11 bits and an extended message is 29 bits, the quickest possible rate at which complete messages can be sent is a function of the smallest message and the quickest CAN transmission rate (in bits/s)

Another consideration for the size of message is that there is a gap in transmission between the end of one message and the start of another (typically 3 bits) [71] and the practice of ‘bit stuffing’ which adds extra bits to the message when any five consecutive bits are the same. That is, if five consecutive zeros are sent in the entire transmission, they are followed by an automatically inserted non-zero so as to assist with synchronisation of the CAN nodes. A standard transmitted package is 44 bits + 8n message bits (n bytes of 8 bits each). Extended message transmissions contain 44 + 18 +8n

Bit stuffing may be enacted on 34 of the 44 bits leading to a maximum number of bits after bit stuffing of

44 + 8n + (34+8n-1)/5 total possible bits in the complete frame

62 + 8n + (52+8n-1)/5 total possible bits in the complete frame.

Therefore the maximum size of any frame is 128 bits for the 11 bit message ID frame and 149 for the 29 bit message ID [15]. For data consisting of only one byte, the minimum message length is without stuff bits and is 52 bits in total. For the extended message ID, the minimum size of any message is 70 bits. So the highest rate at which any complete frames (packets) can be transmitted, assuming no stuff bits and minimum message data, is

Table 4-1 : maximum numbers of messages that can be sent using different CAN speeds

	Message size			
	11 bit msg ID		29 bit msg ID	
CAN speed (bits/second)	52	128	70	149
125,000	2404	977	1786	839
250,000	4808	1953	3571	1678
500,000	9615	3906	7143	3356
1,000,000	19231	7813	14286	6711

Table 4-2 : Number of messages and average frames per second that can be sent using different CAN speeds

	Message size				
	11 bit msg ID		29 bit msg ID		
CAN speed (bits/second)	52	128	70	149	Average frames per second by CAN speed
125,000	2404	977	1786	839	1501
250,000	4808	1953	3571	1678	3003
500,000	9615	3906	7143	3356	6005
1,000,000	19231	7813	14286	6711	12010
Average frames per second by msg size	9014	3662	6696	3146	

Table 4-3 : maximum numbers of messages that can be sent using different CAN speeds for given numbers of nodes

	Msg size			
	11 bit msg ID		29 bit msg ID	
No of Nodes	52	128	70	149
5	481	195	357	168
10	240	98	179	84
50	48	20	36	17
80	30	12	22	10
100	24	10	18	8
120	20	8	15	7

4.5.2 Sample rates and message frequency

For a receiving node to be able to detect, read and process an incoming message from a single sending node, the rate at which the receiving node samples the incoming messages and is ready to receive another, must be at least equal to the rate at which the messages are being sent. Otherwise, an accumulation of buffered and unread messages will cause the receiving node to either fail in reaching the end of the queue of buffered messages, or the buffer will refresh and overwrite data that has not had time to be read. The second of these is more desirable in that the most recently read message will be up to date, albeit that another more important message may have been completely discarded without being acted upon. In the case of messages filling the buffer at too fast a rate for the receiving node to read them, the receiving node will eventually be able to read only old potentially out of date messages and important, high priority, safety related messages could be missed completely or only read after they have missed some deadline. For example, if a change in the speed of the wheels is not communicated to the ABS controller, a serious safety issue will ensue in which the brakes could be held on during a skid or be released when good braking is possible but not demanded by the ABS controller. The ability to read as many messages as are sent is therefore critical.

The maximum number of CAN messages per second is 8771 with no contention and a bit rate of 1Mbit/s [15]. In experiments with two nodes transmitting using CAN at 125kbit/s, in which one sends a sequence of messages separated by an arbitrary time interval, messages sent with a delay of as little as 10 milliseconds can be received by a single node and echoed back after some processing to populate the message send buffer before the next message has been sent.

With no time delay between messages in the sending node, the receiving node fails to correctly receive and echo all of the sent messages in a timely fashion despite the echoed message having a higher priority (lower ID number) than any message that is attempting to be sent by the sending node. When the separation of messages is only 10 milliseconds, the load on the bus is measured at 20% by the ADFweb CAN analyser. With no programmed delay between the messages, a natural time interval of almost zero to 2 milliseconds because

of processing time on the Arduino CAN shields means that messages are sent on average every 1 millisecond (1000 messages per second). At a baud rate of 125kbit/s this exceeds the potential of the bus and overloads it to the point where every message (excluding the first one sent) is competing in arbitration with the next or it is waiting for the bus to become idle before attempting to send.

Even with as little as a 1 millisecond delay (when only one node is transmitting) the bus load is still only 21% and with a 10 millisecond delay this falls to 3% for the single node model and a message time of 0.25 milliseconds at 500Kbit/s baud rate. With a delay of only 1 millisecond, the sent message can be received, stored, packaged and returned within 0.65 milliseconds with a bus load of 42% at 500Kbit/s. By changing nothing else, but decreasing the CAN speed to 125Kbit/s, the bus load increases to 100%.

4.5.3 Issues with the resolution of vehicle speed data transmitted using CAN

In the discussion of the CAN protocol in the BMW Mini implementation the unsigned 16 bit value type can represent decimal fractions of the wheel speeds in steps of 1/64 or 0.015625, which is a problem when the physics model attempts to return values with higher resolution or greater precision than the 1/64 achievable with an integer value that undergoes conversion to and from a floating point number during calculations in the physics model.

If, for example, an input speed is transmitted via CAN, it will first be converted to a value that can be stored in 16 bits as an integer. The largest value that can be stored in this type is 65535 and, with the maximum quoted wheel speed of 100 m/s, a scaling factor of 1/640 would allow this to fit into a 16 bit unsigned int as the value 64000 or 1111101000000000

The next possible speed that could be represented in this way would be the binary representation 1111100111111111 = 63999 and dividing this by 640 gives 99.99844 or 100-(1/640) which is already an improvement on the precision of the 1/64 that the BMW Mini implementation allows for, presuming it does indeed encode an integer to be transmitted as 16 bits via CAN.

Alternatively, the BMW Mini could be using integers in the range of 0 to 6400 to represent a maximum resolution of 1/64 from 0 to 100 m/s but this seems wasteful of the 16 bit resource available for the wheel speed. It could be that the Mini encodes a short floating point type and passes this as 16 bits without conversion but retaining a maximum precision of 1/64 for reasons that suit their processors' arithmetic logic unit (ALU) and to avoid conversions on both sides of the CAN message transmission and reception.

In the Arduino simulation, the physics model makes calculations on float and double types before converting to an integer to be passed as a 16 bit 'byte pair'. This causes errors of truncation and loss of accuracy when values less than 1/64 appear in the decimal fraction portion of the values calculated. This is tolerable in the simulation considering the actual implementation on a vehicle would presumably have sensors that can only send values in increments of 1/64 and calculations to reflect this could be programmed in to the physics

model. Whilst this would be relative easy to achieve in software for the simulation, consideration must be given to the processing time involved in these conversions and calculations and the effect they would have on timings for the transmission of messages that need to meet deadlines. Whilst a non-real-time simulation would simply take this into account in the results provided against the simulated timings, a real-time (non-HILs) model could suffer from delays in processing that would result in the model not being able to run at real time speeds – even though capable of calculating the correct physical values.

4.6 Network Topologies

Networks of ECUs in modern vehicles can be connected in many different ways, from point to point connections (where every node can communicate directly with any and every other node in the network) to star networks and buses (where messages are sent to a shared central hub or to a shared electrical wire before they are received by the node for which they are intended). The number of unique ways in which any number of nodes in a network can be physically connected is the total number of integer partitions of the number of nodes. Constraints on the number of connections in vehicles may be determined by issues of separation of domains, for security for example.

4.6.1 Integer partitions of network configurations

Considering the possible number of candidate architectures with respect to the communications buses, the ‘partitions’ of integers give the total number of different network topologies that can be used for any given number of features of the vehicle.

For example, if there were four features in the ABS case study, the partitions of four are given by $p(n) = p(4) = 5$

{4}
{3, 1}
{2, 2}
{2, 1, 1}
{1, 1, 1, 1}

where each row is a representation of the possible number of features per ECU. The orders of each element of the partitions are not considered in this example and therefore the decision about which features are grouped together on which ECUs is not made by this sequence, that is one candidate architecture for four features may have three features on one ECU and just one on another but nothing is stated about which three or which one.

Similarly, the same system may be made of three ECUs, where one supports two features and two more ECUs support one feature each.

If the system specifies five features, the number of partitions increases to p(5)=7 and for ten vehicle features, p(n) = p(10) = 42.

Whilst these numbers are manageable and do not take long to compute and display by a relatively short algorithm, it doesn't take long for small increases in n to result in unmanageable numbers of partitions, for example p(50) = 204226 and a more likely calculation based on the number of features/ECUs in current vehicles p(100) = 190569292

If the calculation and printout of all of the partitions of n = 100 took one tenth of a second each, it would take more than seven months to print the entire number of partitions as single rows, like in the example for p(4). If the order and specific groupings of features that share the same ECUs is taken into account, permutations and combinations of the possible partitions need to be calculated. For example, all four (out of four features) on a single ECU can only be achieved in one way as can four separate ECUS with one feature on each ECU. For the case of an ECU sharing two features, it may be that features a and b are shared and both c and d are on one ECU each. Equally possible, b and c may share the same ECU whilst a and d each have a single ECU. The total possible combinations for sharing any of the features in the same proportions as the partitions of four features is

{4} = {a,b,c,d}

{3, 1} =

{a,b,c}{d}
{a,b,d}{c}
{a,c,d}{b}
{b,c,d}{a}

{2, 2}=

{a,b}{c,d}
{a,c}{b,d}
{a,d}{b,c}

{2, 1, 1}=

{a,b}{c}{d}
{a,c}{b}{d}
{a,d}{b}{d}
{b,c}{a}{d}
{b,d}{a}{c}
{c,d}{a}{b}

{1, 1, 1, 1}=

{a}{b}{c}{d}

Therefore, the total number of ways that four features can be partitioned so as to either share one ECU or to be completely separated on to four ECUs and every possibility in between is a function of the number of different partitions and the combinations within each element of the partition.

For example, in the case of {2, 1, 1} above, calculating C(4,2) gives the total number of possible combinations of the {2, 1, 1} partition, because what's left after the number of ways of placing two of the elements in one set is fixed. That is, we are only interested in the makeup of the 2 elements that are paired together, for example {A B}, {A C}, {A D}, {B C}, {B D}, {C D} and what remains outside of any pairing is one distinct pair that can be separated into two singletons.

For the case of the {2, 2} partition, after choosing the six distinct sets shown above in the {2, 1, 1} case, the remaining selections that could pair with the chosen six are in fact the same six sets repeated which has the effect of halving the number of pairs of sets of two that make a unique superset.

In Figure 4-18, ECUs are assigned to support a single brake pedal, an ABS controller, an IMU, four brake callipers, four wheel speed sensors and a buffer to create a single data field which can be transmitted via CAN with a single message ID.

In the Bosch manual for their M4 Motorsport ABS, the four wheel-speed values are sent to the CAN bus in a single data field with just one shared message ID (hexadecimal 0x24A). In this author's ABS model, in order for the four wheel speeds to be brought together into the single data field, a separate 'wheel speed collector' listens on the bus and gathers updated values from each wheel sensor's ECU individually before sending the values back to the bus for the ABS controller and whichever other nodes may require them.

Alternatively, the same effect can be achieved by having a dedicated bus, solely for the wheel speeds but not necessarily CAN, on which a fifth ECU receives each individual speed value and sends a single message to the CAN bus or a single ECU is fed the wheel speeds on four separate GPIO pins before they are collated and sent to the CAN bus.

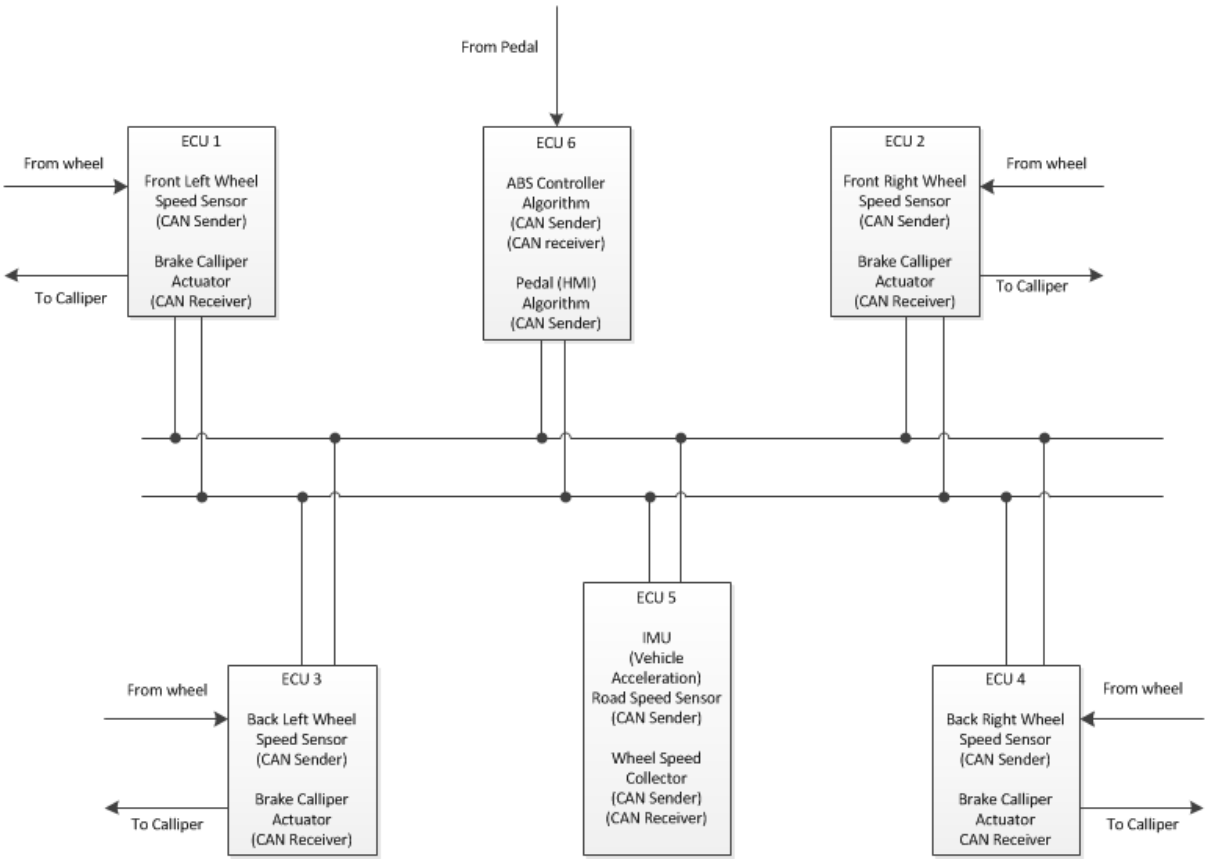


Figure 4-18: Author's interpretation of shared ECUs supporting multiple features on a single communications bus

4.6.2 Vehicle Domains

Vehicle domains are the areas into which the uses of ECUs are divided. Traditionally they are powertrain, chassis, comfort, infotainment, safety, ADAS, cockpit, security and others depending on the vehicle's original equipment manufacturer (OEM).

A typical arrangement of vehicle domains will have each domain as an individual node in a network, separated from the others for processing of normal operations (the control algorithms) and connected for communication by a gateway at each node. Separation is necessary if there is any reason that one domain could have a negative safety or security implication on another domain. For example, a window winding controller that can communicate with the safety or security domain to allow correct operation of the alarm or immobiliser could also be a point of vulnerability from an attack or hacking. Whilst a single ECU with just one processor controlling both could be problematic without the use of a hypervisor, allowing separate domains to communicate can be controlled so that only valid messages can be received and acted upon. In Figure 4-19, eight domains are connected via a bus and/or protocol that allows virtual point to point communication via a central hub or by a shared twisted pair such as CAN.

Katzan [72]describes a hypervisor as a control program that, along with a special hardware feature, permits two operating systems to share a common computing system. A relatively small hypervisor control program is required which interfaces the two systems.

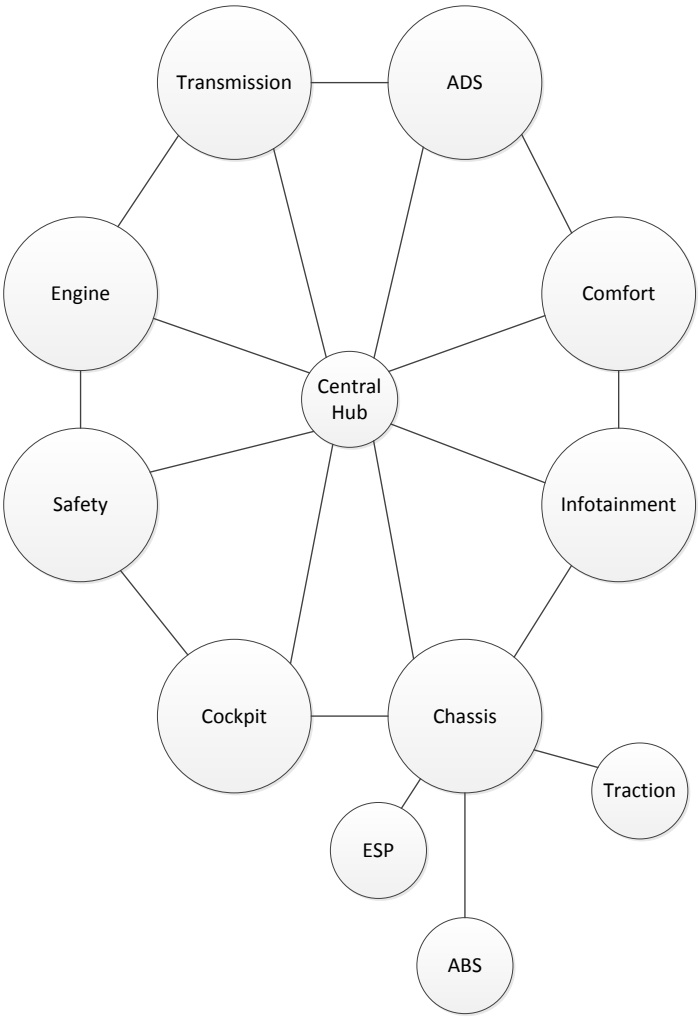


Figure 4-19 : Example network of eight domains with the ABS controller, Electronic Stability Program and Traction belonging to ‘Chassis’

Each component of the chassis domain is a node in the chassis network. The proposed optimisation process delivers a candidate solution for the topology of the ABS in isolation, without concern for either Electronic Stability Program or traction-control. When a solution has been found to the ABS hardware/software architecture, the GA can be run against any other system at the same level of the system hierarchy. Having found solutions for each of the subsystems of the chassis domain, the GA or a version of it can be run to find a solution to the integration of each domain as a complete system within the vehicle. One of the nodes, or a separate bus controller, acts as a gateway to allow the communication between chassis and other domains. If other domains use a different protocol, an interface is required to integrate the two domains.

4.7 Results & Discussion

For the specific ABS problem tackled by this research in creating a system of ECUs on Arduino boards and a CAN bus, executable models created in software to simulate each feature of the vehicle are shown in table 4-4 to have their own requirements for program storage space, determined by the C compiler for the Arduino experimental boards when each individual feature was assigned a single ECU. This is the major criterion for allocation of ECUs to features and both the GA and bin packing algorithms take account of these in reaching candidate solutions.

Table 4-4 : Memory and processor speed requirements for the ABS components

ID	Name	Program storage space (bytes)	Local variables (bytes)	Processor speed (Hz)
0	"Pedal "	6306	289	4000
1	"ABS "	8910	305	4000
2	"Physics "	14962	551	4000
3	"IMU "	7264	245	4000
4	"Wheels "	7576	277	4000
5	"Callipers "	5657	259	4000
6	"Environment "	4314	222	4000

The known properties of the Arduino boards, also output at compile time or taken from the datasheet (and confirmed by visually reading off from the clock chip), were used to produce ECU data input for the GA and bin packing executables in table 4-5. For the specific features of the ABS system, a maximum of six ECUs' data was input to the GA (albeit they are identical in this case). An additional field, for cost to purchase for each ECU, was included but is not shown here as it is not a constraint on capacity for any property of the features and they are identical for these specific ECUs).

Table 4-5 : Memory and processor speed properties of the ABS ECUs

ID	Name	Program storage space (bytes)	Local variables (bytes)	Processor speed (Hz)
0	"ECU 0 "	32256	2048	16000000
1	"ECU 1 "	32256	2048	16000000
2	"ECU 2 "	32256	2048	16000000
3	"ECU 3 "	32256	2048	16000000
4	"ECU 4 "	32256	2048	16000000
5	"ECU 5 "	32256	2048	16000000
6	"ECU 6 "	32256	2048	16000000

Two sets of results were obtained from the GA. The first, from an exhaustive run across all 7^7 possible solutions, produced the candidate solution

ECU 0 supports features : 3 4 5 6
ECU 1 supports features : 0 1 2

in 2.5 seconds where all ECUs have the same properties equating to a solution where the total search space is mapped as in table 4-6

Table 4-6 : Results of ESP for ABS with 7 nodes

ECU/Feat	0	1	2	3	4	5	6
0				x	x	x	x
1	x	x	x				

ECU 0 supports : IMU, wheels, callipers, environment
ECU 1 supports : Pedal, ABS, Physics model

The Bin packing solution for the same problem (without ordering of features) yields the solution

ECU 0 supports features : 0 1 2
ECU 1 supports features : 3 4 5 6

in 0.008 seconds equating to a solution where the total search space is mapped as in table 4-7 and because of the similarity of the ECUs, this is actually the same result as the ESP.

Table 4-7 : Results of Bin Packing algorithm for ABS with 7 nodes [0001111]

ECU/Feat	0	1	2	3	4	5	6
0	x	x	x				
1				x	x	x	x

This result shows that the bin backing algorithm produces an equally good solution as the one from the ESP in just 2/625ths of the time for this specific problem.

Taking only 0.0028 seconds to produce the following in a maximum of 2^8 iterations of which 68 were executed, ten runs of the GA produced four solutions that used only two ECUs, as in table 4-8 to table.

Table 4-8: Results of GA for ABS with 7 nodes [0055050]

ECU/Feat	0	1	2	3	4	5	6
0	x	x			x		x
5			x	x		x	

Table 4-9: Results of GA for ABS with 7 nodes [0111000]

ECU/Feat	0	1	2	3	4	5	6
0	x				x	x	x
1		x	x	x			

Table 4-10: Results of GA for ABS with 7 nodes [2424424]

ECU/Feat	0	1	2	3	4	5	6
2	x		x			x	
4		x		x	x		x

Table 4-11: Results of GA for ABS with 7 nodes [1100110]

ECU/Feat	0	1	2	3	4	5	6
0			x	x			x
1	x	x			x	x	

The hardware/software CAN network on Arduino experimental boards performed within expected parameters and demonstrated a fully functioning ABS model with the ability to simulate a vehicle being driven at speed and then slowed to a halt by the application of a simulated brake pedal that interacts with the ABS controller and ultimately the brake callipers via CAN messages. The application of the brake pedal was simulated in figure 4-20 with minimal pressure required to cause some retardation and again in figure 4-21 with the pedal fully depressed.

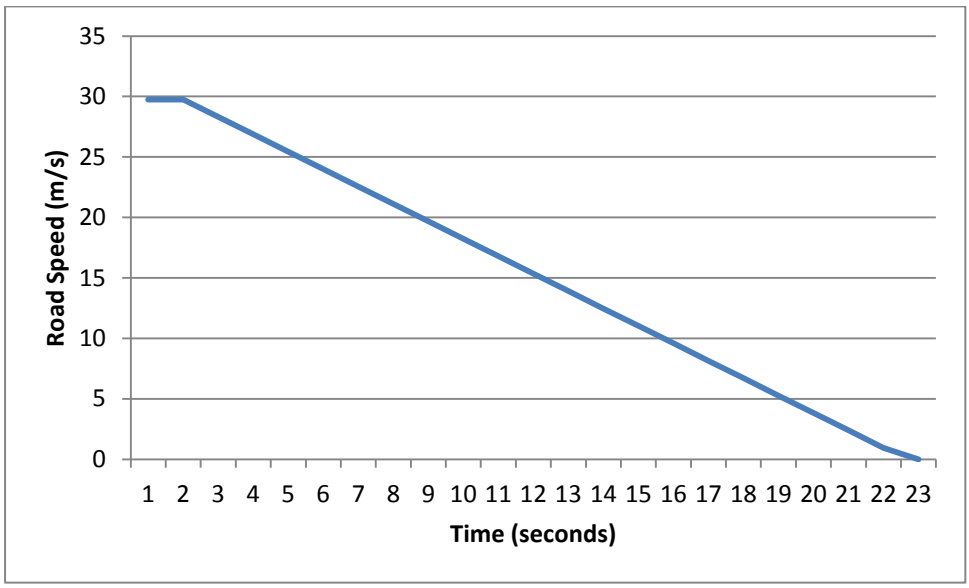


Figure 4-20 : Braking from 30m/s to full stop with minimal brake pedal pressure

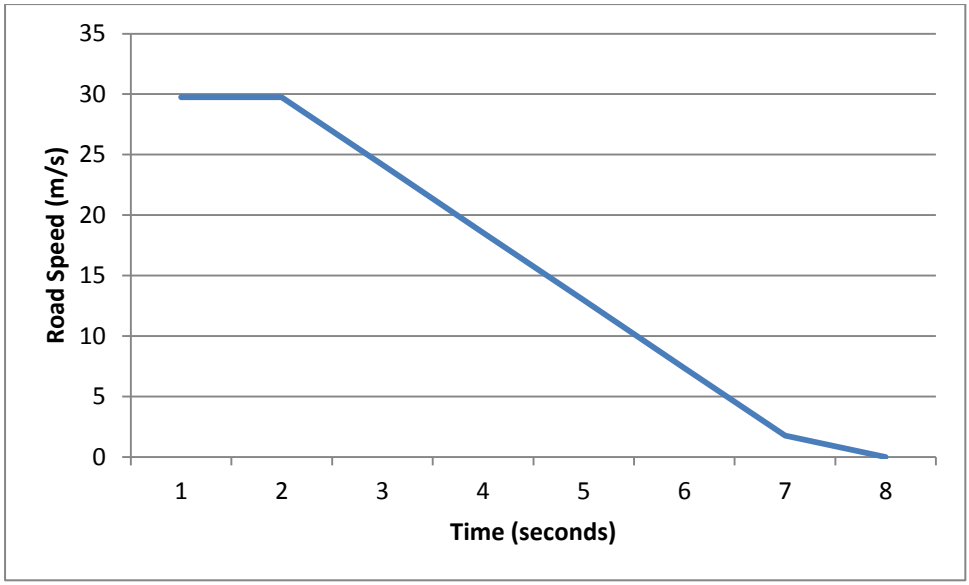


Figure 4-21 : Braking from 30m/s to full stop with increased brake pedal pressure

4.8 Conclusions

The ABS case study showed that a separate controller and dynamical simulation model could be designed and integrated using rapid prototyping methods such as a diagrammatic coding design tool and a genetic optimisation algorithm. The components of the ABS system were modelled in UML and limited code was automatically generated before manual intervention was required to complete the coding task for the design of ECU software for each individual component of the ABS system, enabling the support of each feature on its own ECU with communications protocol driver software provided, ready to use, by a third party. The programming of the ABS control software and the prioritised messaging algorithms for a CAN bus were conducted as part of this research and were implemented on readily available μC boards using the C programming language and the MCP2515 CAN chip. The design tool that this research developed and employed was validated by the hardware/software architecture deployed on a distributed network of ECUs communicating via CAN

The ABS case study was limited in scope and it would be beneficial for further research to be carried out with emphasis on obtaining or developing better physics models and ABS control algorithms. Access to different hardware, apart from the Arduino boards and Seeed shields would allow a wider examination of potential candidate solutions with options for exploring the benefits or otherwise of alternative communications protocols.

5. Integrating ABS with Automated Manual Transmission (AMT)

To verify the optimisation and distributed system design tool developed in this research, a second case study, "Integrating ABS with Automated Manual Transmission (AMT)" considered the design of AMT control software following the same design processes as the ABS system and used both the bin packing and the genetic algorithm to verify an architecture that could be deployed in a vehicle with AMT alone and with an ABS/AMT model integrating the two and adapting the physics model to simulate both. A candidate solution was produced for the AMT in isolation from the ABS or any other system of the vehicle. This solution can now be used in conjunction with any other previously obtained solutions for other systems in a further execution of the GA and bin packing models where each solved system becomes a node in a new problem or the two systems become one with their respective software for each individual node shared on even fewer ECUs.

AMT is not the oxymoron it might sound at first. AMT uses either a standard 'H-pattern' gearbox or a sequential box similar to those used on motorcycles but performs the mechanics of the gear changes (up and down) automatically upon the request of the driver, who presses a switch either hidden inside a floor mounted gear lever or by paddles on either side of the steering wheel. This is commonly known as "flappy paddle shift" gears. The software only needs to receive input that informs whether an upshift or a downshift is called for, hence the usefulness of a sequential box, since to get from say fourth gear to second gear third must be engaged first unlike in a completely manual 'H-pattern' box where the driver may miss out gears when going either up or down.

The algorithm controls the engagement and disengagement of the clutch (so there is no clutch pedal in vehicles with AMT) and matches the engine speed to the road speed during downshifts by controlling the throttle while the gears and the clutch are both disengaged. This reduces wear on the synchromesh mechanisms as both halves of the synchromesh are moving at the same speed when they engage.

This research treats AMT in the same way as ABS, considering the physical components of the system and assigning the software control algorithm for each of these to a single ECU to be treated as individual features supported on one ECU each before applying the GA to reduce the number of ECUs used to support the AMT system.

An AMT controller feature that manages smooth and appropriate gear changes is employed much like the ABS controller and a physics model dedicated to the AMT exists in isolation before being integrated with the ABS physics model to reflect vehicle dynamics affected by both AMT and ABS.

5.1 Components of AMT

AMT requires the usual components of a manual gearbox and a selector mechanism for use by the driver. For easy operation of gear selections, the box is usually sequential like those normally associated with motor cycles, so that the gears can be selected and engaged with a

push or pull of a mechanical lever that can only move in a forward (up) or backward (down) direction. This allows a much easier automation of the movement of the selector lever than would be possible with an H-pattern gearbox. To allow for automation in conjunction with manual gear selection by the driver, the gears are moved into position by an electric motor or solenoid and this is activated by outputs from the AMT controller after it has processed requests for gear changes made by the driver. If a request meets the criteria for an appropriate gear change, the physical selection of the gear will be executed. Therefore, the components that are considered in this research to be part of an AMT system are

The gear-stick (optionally flappy-paddles) to transmit 'up/down' messages

Accelerator pedal (position value transmitted to CAN)

Engine (Revs transmitted to CAN)

IMU (vehicle speeds will determine whether gear selection is appropriate)

AMT controller (like the ABS controller, manages AMT)

Gearbox (actuator physically engages gears, numbers and ratios)

Clutch actuator (receives CAN message (open/closed) from AMT controller)

5.2 Appropriate gears and the AMT controller

Having established the components that make up the AMT system, it is useful to consider how each of them is processed by the AMT controller. The AMT manages appropriate gear changes by measuring the current speed and engine revs together with any requests for up or down shifts and allows the gear change or not depending on whether sufficient road speed has been achieved for up shifts or if engaging the gear would damage the engine by over revving if, for example, the road speed is too high for the requested gear. Table 5-1 shows a range of appropriate road speeds for up and down gear changes based on vehicle data for gear ratios in a Chevrolet Corvette Stingray – 2014 available online at [73].

Table 5-1 : Road speed for given gear selections

Gear		Revs Range		Speeds Range	
From	To	Min	Max	Min	Max
1	2	929	6503	5	23
2	3	929	6503	6	36
3	4	929	6503	8	38
4	5	929	6503	12	53
5	4	929	6503	12	53
4	3	929	6503	8	38
3	2	929	6503	6	36
2	1	929	6503	5	23

For example, figure 5-1 shows the road speeds achieved in each gear for the rev bands between 929 and 6503 (nominally 1000 revs to 6500 revs). The axis labels, "Gears x 10,000 + Revs" refers to a concatenation of the gear number (from 1 to 5) and the revs in that gear so that for example 11000 is gear number 1 followed by 1000 revs. Each new gear starts at 1000 revs (this is assumed to be the tick over value of the engine below which it would be difficult or impossible to drive without stalling the engine). Similarly, in Figure 5-2, the values on the *x* axis are a concatenation of gear number and revs. The same information is shown in both figures with smooth reductions in speeds between gear changes or with stepped values.

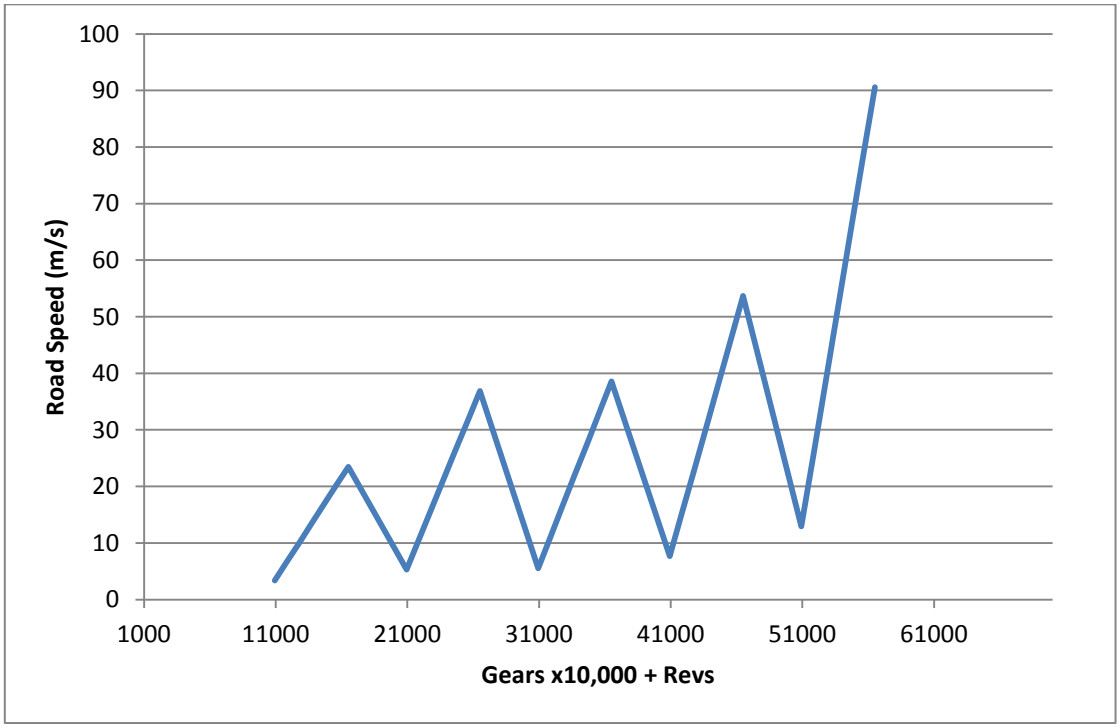


Figure 5-1 : Graph of road speeds by steps of 929 revs in each gear

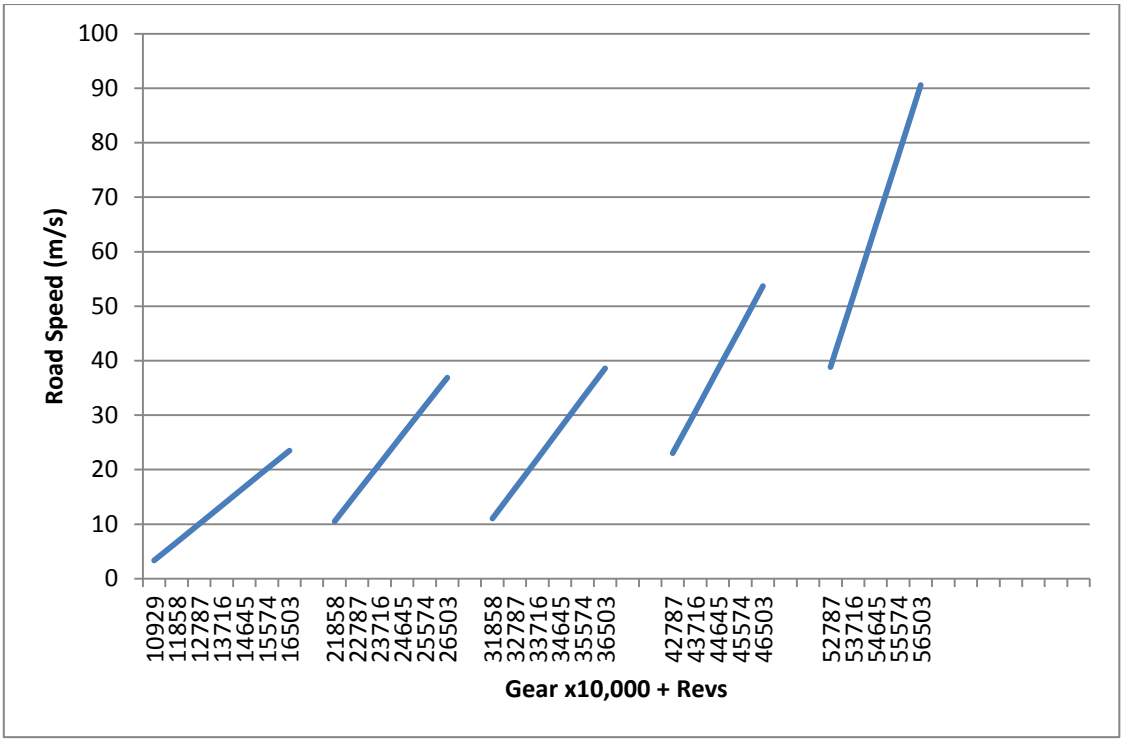


Figure 5-2 : Graph of road speeds in mph by steps of 929 revs for each gear

Figure 5-3 shows the ranges of speeds in each gear with markers to show the speeds at intervals equating to multiples of 929 revs.

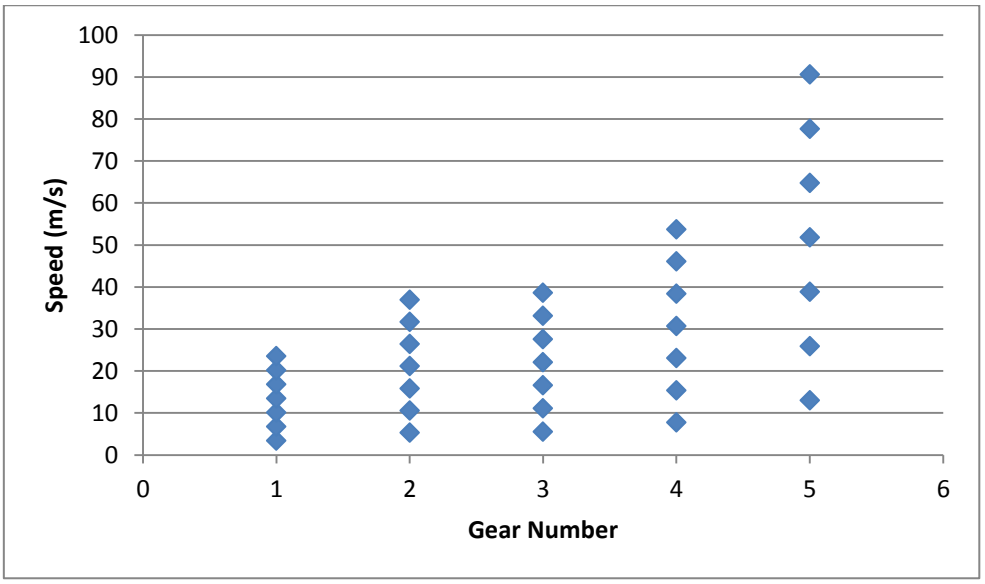


Figure 5-3 : Ranges of road speeds in gears 1 to 5

An algorithm that can determine the correct gear based on the speed of the vehicle and the engine revs would test for change-points. These could automatically make gear changes or trigger warnings to the driver to change up or down manually.

The algorithms for the AMT controller are concerned with managing gear changes as requested by the driver. The driver selects the desired gear by selecting the next gear up or

down from the current gear in a sequential list. That is, if the car is currently engaged in 1st gear and the driver wants to move into 2nd gear, this is done by the driver pushing or pulling on a stick or paddle to indicate a request for an ‘upshift’. This is the exact same operation when the driver wants to go from 2nd to 3rd, 3rd to 4th etc. and when a ‘downshift is requested from, say, 5th to 4th, 4th to 3rd etc., the driver pulls on another paddle or pushes the gear stick in the opposite direction from that for the upshift. If conditions are not appropriate for the selected gear shift, for example if the road speed is too great for a downshift to be executed safely and without damaging the engine by over-revving, the software will ignore the driver’s request and perform no gear change. Similarly, if the road speed or engine rev count is insufficient for an upshift, the driver’s request will be ignored. Within a range of safe and appropriate road speeds and engine speeds, the driver is at liberty to request any gear at any time without being overridden by the on-board software. If the vehicle comes to a complete rest or the driver slows significantly without changing gear, the software will intervene by making an automatic downshift so as to prevent stalling the engine. In the event of a complete stop, first gear will be selected and the clutch will be disengaged.

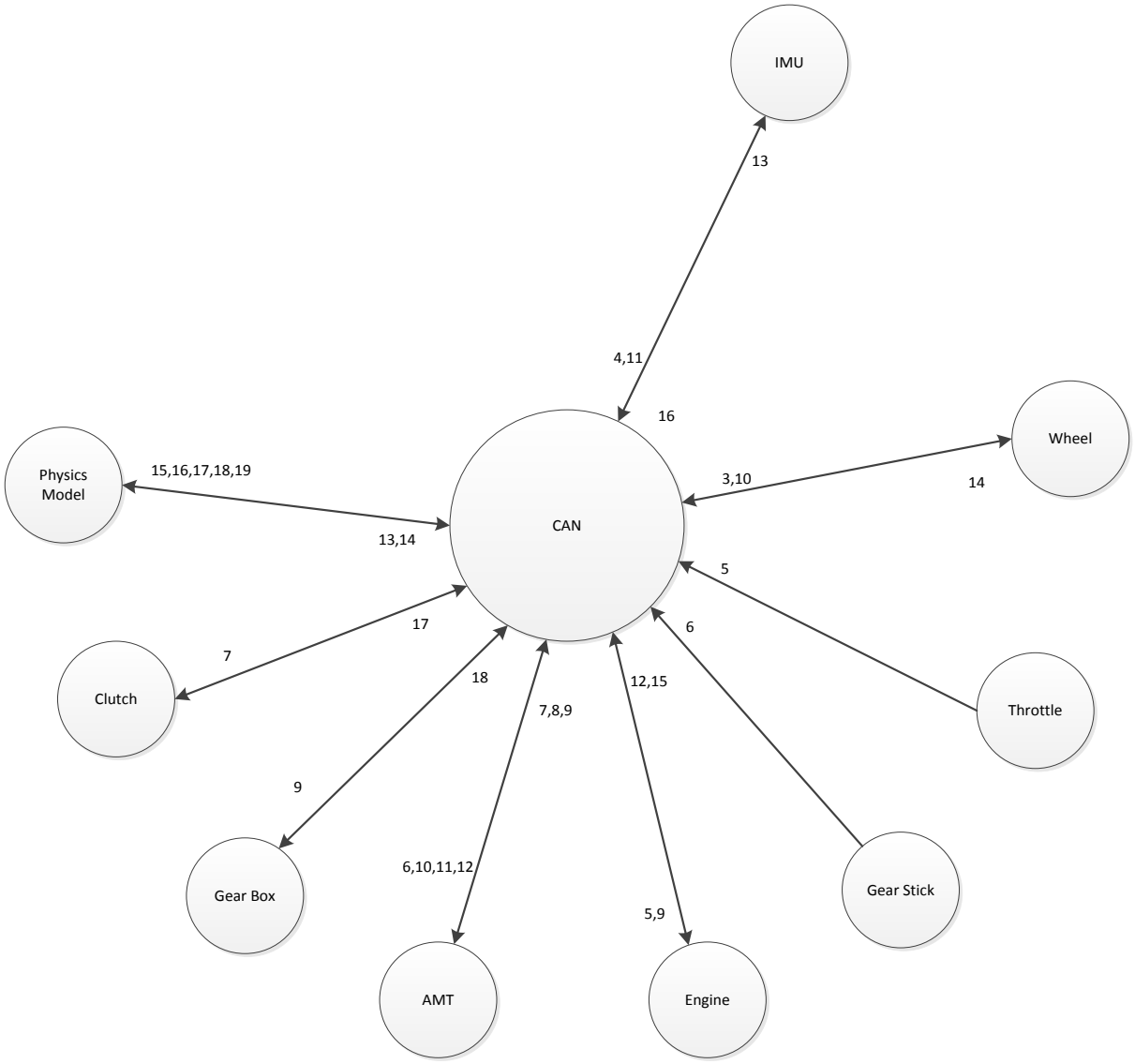


Figure 5-4 : Priorities of CAN messages in an AMT system simulation

5.3 A physical model for AMT

For the control of vehicle speed during acceleration events such as increasing engine revs via throttle control, an alternative physics model was created and tested separately from the physical model's ABS physics function, which had only dealt with the physics of braking. This new AMT physical model was integrated in to the software code for a single physical model that called a separate function to handle the physics of specific braking and ABS events.

CAN messages, conveying the engine revs, vehicle speed, wheel speeds and currently selected gear ratio were sent to the CAN bus via messages broadcasting the IDs 120, 130, 140 and 180 respectively. IDs 120 and 180 were received by the physics model which multiplied the engine revs by the gear ratio to obtain a final drive output value which was then sent directly to the wheel speed sensors' simulator and the vehicle speed sensor's simulator.

Wheel spin was not incorporated in to the physics model and good grip was assumed at all times for acceleration from the throttle pedal. Acceleration was not accurately modelled save to say, for any throttle pedal value that could be achieved, this would translate directly to a vehicle speed completely overcoming inertia in an unrealistic way. Neither the mass of the vehicle nor aerodynamic effects were considered.

As with the ABS physical model, no consideration was given to the materials that make up the components or the tyres or road surfaces. The physical behaviour and deformation of tyres under acceleration events such as moving from standstill or steering are not part of the physical model. The coefficients of friction for each tyre are assumed to be constant with respect to any given road surface or weather condition.

Since the purpose of the research was to demonstrate the potential of a proposed design method for integrated features of a system, distributed across wired networks of hardware/software architectures, less importance was placed on the accuracy of the physical models. These are modular by design and individual models could be swapped out for better ones if further work is carried out on the back of this research.

5.4 Integration of the two systems

AMT and ABS share common features of the vehicle, specifically the IMU and wheel speed sensors. These can be further integrated into an electronic stability program or 'traction control' system. For the AMT, road speed and wheel speeds are matched to engine speeds so that appropriate gear changes can be made whilst still allowing the driver some control over whether to use high revs in low gears for torque and acceleration or to make a more conservative and economical choice of gears with low revs when speed and power are not a priority. During braking events, the gears can be used to provide engine braking and downshifts during braking that uses 'brake demand modulation'. ABS events can be managed, automatically, to provide safe and efficient braking in all conditions without loss of steering control.

Integration of ABS and AMT requires communication between the vehicle domains of 'Transmission', 'Engine' and 'Chassis'. After the solution of the ABS's ECU allocation problem and that of the AMT, a further process offers an optimised integration solution for the ABS and AMT modules, with one node from each sub-network acting as a gateway, to allow communication directly between the two or via a third communication bus.

Figure 5-5 shows an example of the simplest integration of the two systems, each operating separately on their own cluster of ECUs dedicated to the task of ABS and AMT. At this point in the design process, there is no sharing of any of the features from separate domains on a single ECU. Even the shared components of both systems (that is wheels speeds and IMU) are assigned to ABS because this was the first of the two systems to be modelled and this simple integration is just a matter of connecting the two domains via the CAN bus. Subsequent solutions to the integration problem share features from both domains which raises a question of whether the classification of the two systems as belonging to separate

domains is correct. Normally, but not always, domains cannot interact on the same level of communication or on the same ECU for reasons of safety or security. Allowing the infotainment system to reside on the same ECUs as the vehicle alarm and immobiliser systems would be a potentially dangerous sharing of software/hardware for a domain that may be prone to attacks facilitated through the ease with which radio signals might be used to gain access to the security system via the radio, for example. The domains of security and infotainment would usually be kept apart physically or by encryption on the messages, if any, sent and received between the two.

Having designed each of these separately for this research, as would be the case in a fully implemented vehicle, the integration of ABS and AMT can be a further refinement of the ECU/Features problem that can be solved before the production and implementation of the constituent parts of ABS/AMT. Executable programs (or modular source code) for the individual features can be allocated to ECUs in such a way as to allow many different features, from different vehicle domains, to be shared on the same ECU.

Taking all of the features that make up both ABS and AMT, they can be grouped together into a single run of the GA to provide a solution that does not discriminate between the two sets of features and produces candidate solutions capable of sharing the code from both on any one ECU.

For example, each high level feature (AMT and ABS) can be solved separately for the ECU/Features problem and kept as separate entities with no overlap, save for communication between the two via the two buses that serve each and/or with a third bus for both to communicate with. Whilst the entirety of ABS/AMT features could be separated out into a 1:1 mapping of ECU to feature (this is the default solution for every problem that the GA attempts to solve) and treated as a single integral feature or domain, overlap of features from one domain to the other can be averted as in Figure 5-5, or different amounts of overlap ranging from a single feature to the total integration of both systems on to a single ECU.

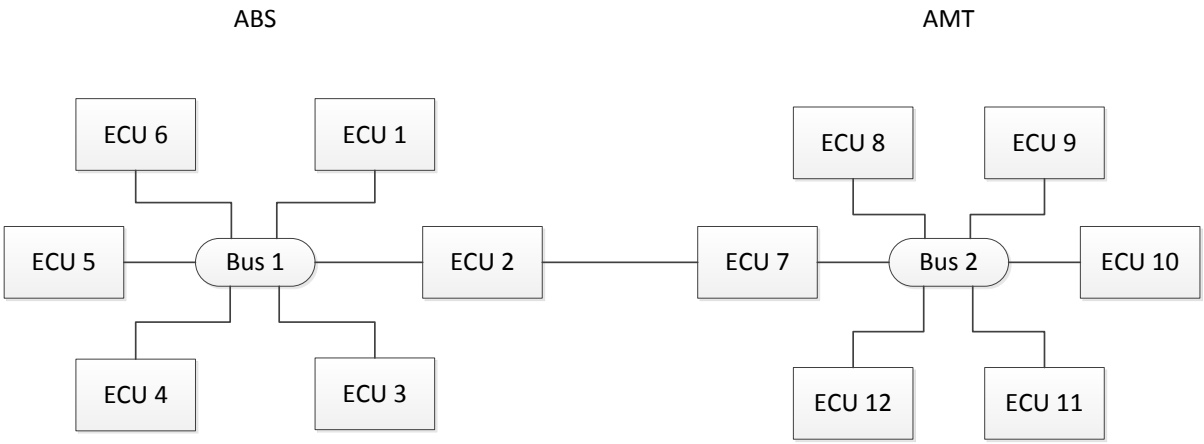


Figure 5-5: Two buses connected by single nodes on each as a gateway

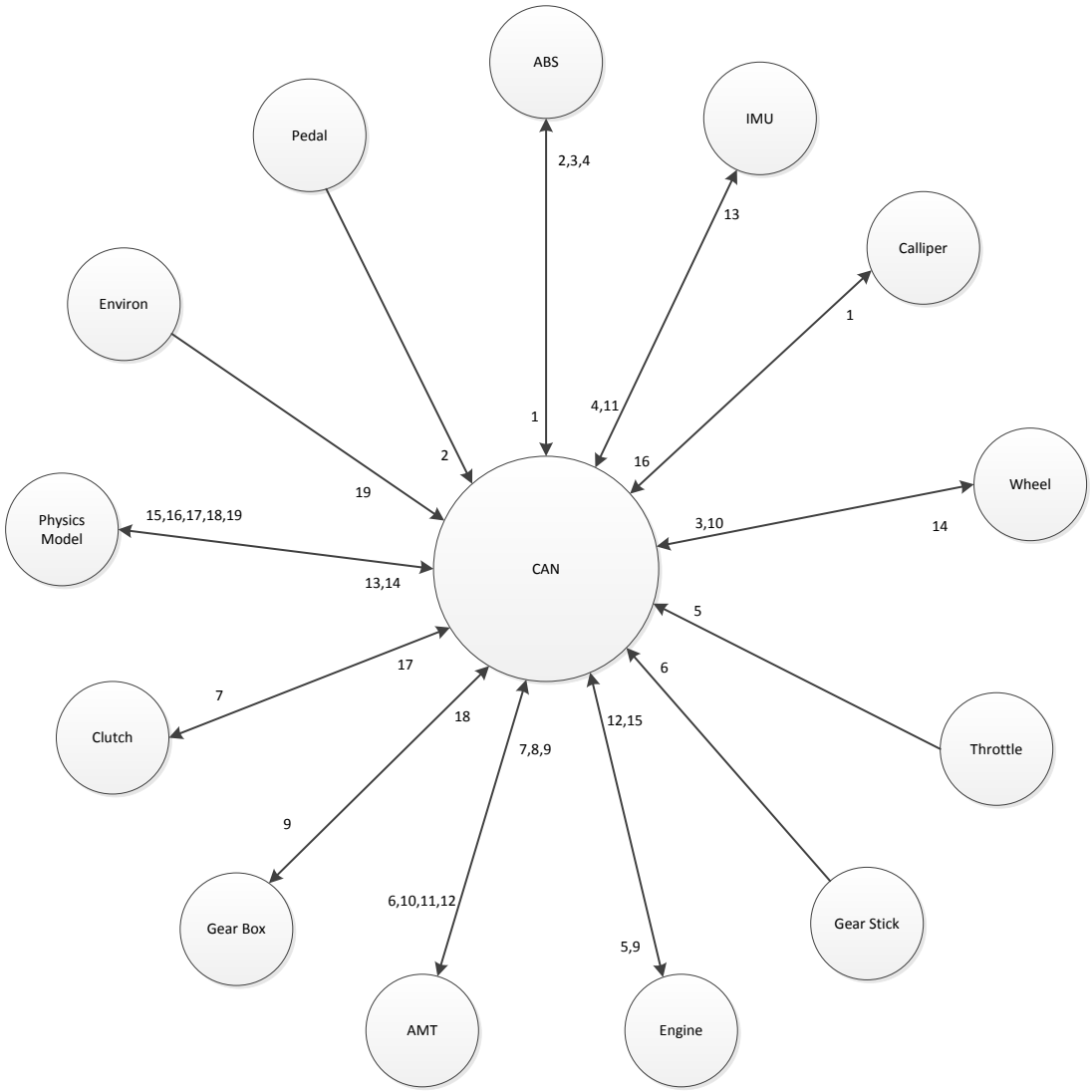


Figure 5-6 : Message priorities in an integrated ABS/AMT ECU/CAN architecture

5.5 Results & Discussion

Similarly to the ABS system, the features of the AMT produce numerical measurements of required memory, shown in table 5-2, when compiled for the Arduino boards and the same ECU data, but for eight ECUs, is provided in table 5-3.

Table 5-2 : Features of the AMT system, including those that it shares with ABS

ID	Name	Program storage space (bytes)	Local variables (bytes)	Processor speed (Hz)
0	"Physics "	14962	551	4000
1	"IMU "	7264	245	4000
2	"Wheels "	7576	277	4000
3	"Throttle "	7692	293	4000
4	"Gear Stick "	4852	214	4000
5	"Engine "	5254	364	4000
6	"Gear Box "	5298	223	4000
7	"AMT "	6294	339	4000

Table 5-3 : Memory and processor speed properties of the AMT ECUs

ID	Name	Program storage space (bytes)	Local variables (bytes)	Processor speed (Hz)
0	"ECU 0 "	32256	2048	16000000
1	"ECU 1 "	32256	2048	16000000
2	"ECU 2 "	32256	2048	16000000
3	"ECU 3 "	32256	2048	16000000
4	"ECU 4 "	32256	2048	16000000
5	"ECU 5 "	32256	2048	16000000
6	"ECU 6 "	32256	2048	16000000
7	"ECU 7 "	32256	2048	16000000

A simple HILs of the AMT was demonstrated to be able to affect the speed of the vehicle with respect to throttle input and gear selection requests with an adaptation of the ABS physics model so as to drive the vehicle up and down the gearbox and through the rev range of the engine.

An integrated ABS/AMT model, with separate ABS and AMT domains, was demonstrated in hardware/software for purposes of validating the system design and verifying the GA. For the AMT model that has eight separate features (not all mutually exclusive with respect to the ABS model) the GA produced solutions comparable with the ESP (3 ECUs used) on 50% of executions. Results from individual runs of the GA for ABS and AMT suggest that individually, the ABS could be implemented on a minimum of 2 ECUs whilst the AMT could be implemented on a minimum of 3. Although this was possible, for the first attempts to validate the hardware/software simulation each feature of both systems was kept to its own ECU. This created some problems because of a limitation on the number of Arduino prototyping boards that could be connected to the same CAN bus without the removal of the 120 ohm terminating resistors built into each CAN shield. It should have been possible to remove them but only by cutting components from the board and this research did not find any benefit in doing so. Instead, the total number of ECUs was reduced by automating HMI features that could themselves be simulated such as the brake pedal, throttle and gear-stick.

The AMT and ABS could then be verified as two separate systems integrated only by connection to the CAN bus.

With the integration of the AMT and ABS systems into a software/hardware simulation, it was possible to reset the normal conditions of cruising at constant speed from the accelerator pedal incorporated into the AMT. Interacting with the ABS system and the physics model, engine revs and individual wheel speeds could be increased and decreased to set up a road speed prior to engaging in a braking event. From the first two experiments using the throttle pedal to set the initial conditions to a road speed of 29.75 m/s, the AMT model was successfully implemented so as to have a direct effect on the ABS model via CAN before a braking event was initiated that caused the physics model to disconnect the throttle from the engine and disallow any increase in engine revs or in wheel speed directly from the powertrain during a braking event.

The physics model is not restricted to acting as the model for only one test system. It was designed and programmed to be reusable and flexible so that the physics equations of any dynamic system of the vehicle could be modelled in isolation and incorporated into an integrated physics model that would operate on events such as acceleration from standstill, acceleration to higher speeds and negative acceleration due to braking as and when called upon. Early physics models used in this research could only simulate braking events but as the AMT case study became integrated in to ABS, the physics model needed to be able to perform the tasks of both. At any time, the physics model can be reprogrammed and extra functionality encoded, whether or not this is required immediately or at a later date.

The first implementation of the integrated ABS/AMT HILs model using a physics model that had been updated to work with either or both was able to successfully produce real-time updates of vehicle speeds by application of the throttle HMI to increase the engine revs and affect the speed of the driven wheels to accelerate the simulated vehicle from zero. A simulated speed of just below 30 m/s was reached. Having achieved this by the re-calculation of wheel speeds in the physics model, the ABS function of the integrated case study successfully performed when the brake pedal HMI was activated. The simulated vehicle slowed to a full stop in accordance with the dynamics equations encoded into the physics model. For the specific AMT problem tackled by this research in creating a system of ECUs on Arduino boards and a CAN bus, two sets of results were obtained from the GA. The first, from an exhaustive run across all 8^8 possible solutions, produced the result

ECU 0 supports features: 5 6 7
ECU 1 supports features: 2 3 4
ECU 2 supports features: 0 1

in 59.1 seconds and where all ECUs have the same properties equating to a solution where the total search space is mapped as in table 5-4

Table 5-4 : Results of ESP for AMT with 8 nodes

ECU/Feat	0	1	2	3	4	5	6	7
0						x	x	x
1			x	x	x			
2	x	x						

ECU 0 supports : Engine, Gearbox, AMT
ECU 1 supports : Wheels, Throttle, Gearstick
ECU 2 supports : Physics-model, IMU

The Bin packing solution for the same problem (without ordering of features) yields the solution

ECU 0 supports features : 0 1 5
ECU 1 supports features : 2 3 4
ECU 2 supports features : 6 7

in 0.01 seconds equating to a solution where the total search space is mapped as in Table 5-5

Table 5-5 : Results of Bin Packing algorithm for AMT with 8 nodes

ECU/Feat	0	1	2	3	4	5	6	7
0	x	x				x		
1			x	x	x			
2							x	x

ECU 0 supports : Physics-model, IMU, Engine
ECU 1 supports : Wheels, Throttle, Gearstick
ECU 2 supports : Gearbox, AMT

For the AMT model that has eight separate features (not all mutually exclusive with respect to the ABS model) the GA produced solutions comparable with the ESP (3 ECUs used) on 70% of executions. Taking 1.08 seconds to produce the following, in a maximum of 2^{18} iterations, of which 117 were needed, ten runs of the GA produced seven solutions that used only three ECUs.

Table 5-6 : Results of GA for AMT with 8 nodes [00577507]

ECU/Feat	0	1	2	3	4	5	6	7
0	x	x					x	
5			x			x		
7				x	x			x

Table 5-7 : Results of GA for AMT with 8 nodes [36733667]

ECU/Feat	0	1	2	3	4	5	6	7
3	x			x	x			
6		x				x	x	
7			x					x

Table 5-8 : Results of GA for AMT with 8 nodes [06226062]

ECU/Feat	0	1	2	3	4	5	6	7
0	x					x		
2			x	x				x
6		x			x		x	

Table 5-9 : Results of GA for AMT with 8 nodes [02266660]

ECU/Feat	0	1	2	3	4	5	6	7
0	x							x
2		x	x					
6				x	x	x	x	

Table 5-10 : Results of GA for AMT with 8 nodes [00334403]

ECU/Feat	0	1	2	3	4	5	6	7
0	x	x					x	
3			x	x				
4					x	x		x

Table 5-11 : Results of GA for AMT with 8 nodes [00232332]

ECU/Feat	0	1	2	3	4	5	6	7
0	x	x						
2			x		x			x
3				x		x	x	

Table 5-12 : Results of GA for AMT with 8 nodes [41244112]

ECU/Feat	0	1	2	3	4	5	6	7
1			x			x	x	
2		x						x
4	x			x	x			

The bin packing algorithm and GA were executed to obtain candidate solutions to the integrated ABS/AMT problem with 13 features with the features' requirements data supplied as in table 5-13. Firstly, the bin packing algorithm was run with the features in an arbitrary

(non-sorted) order and produced the output in table 5-14. It was run again after the features were manually ordered by ROM requirement and the output in table 5-15 was produced.

Table 5-13 : Requirements to support the 13 features of the integrated ABS/AMT system

ID	Name	Program storage space (bytes)	Local variables (bytes)	Processor speed (Hz)
0	"Pedal "	6306	289	4000
1	"ABS "	8910	305	4000
2	"Physics "	14962	551	4000
3	"IMU "	7264	245	4000
4	"Wheels "	7576	277	4000
5	"Callipers "	5657	259	4000
6	"Environment "	4314	222	4000
7	"Throttle "	7692	293	4000
8	"Gear Stick "	4852	214	4000
9	"Engine "	5254	364	4000
10	"Clutch "	5482	226	4000
11	"Gear Box "	5298	223	4000
12	"AMT "	6294	339	4000

With 13 identical ECUs available to support the features of the integrated ABS/AMT system, a solution on three ECUs was indeed offered by the bin packing algorithm. A system of four ECUs supporting the controller software for the features of both ABS and AMT produced the output in the appendices from Appendix J to Appendix M when the Arduino HILs was executed. Lines of output from the GA that produced the four ECUs solution are listed here and the allocation of ECUs is set out in table 5-16.

Table 5-14 : Results of Bin Packing algorithm for AMT with 13 nodes (non-ordered items)

ECU/Feat	0	1	2	3	4	5	6	7	8	9	10	11	12
0	x	x	x										
1				x	x	x	x						x
2								x	x	x	x	x	

ECU 0 : Physics : ABS : Throttle
ECU 1 : Wheel : IMU : Pedal : AMT : Environment
ECU 2 : Calliper : Clutch : Gear Box : Engine : Gear Stick : ABS

Table 5-15 : Results of Bin Packing algorithm for AMT with 13 nodes (ordered items)

ECU/Feat	0	1	2	3	4	5	6	7	8	9	10	11	12
0	x	x	x										
1				x	x	x	x	x					
2									x	x	x	x	x

ECU 0 : Physics : Wheel : IMU
ECU 1 : Environment : AMT : Throttle : Clutch : Engine
ECU 2 : Gear Box : Gear Stick : ABS : Pedal : Callipers

Since the integrated system could clearly be supported on just three ECUs and with four HMIs needing to be supported, the GA was executed with one of the HMIs (environment) removed along with the physics model, which this research has separated on to its own ECU, away from the control system on the vehicle. Since the environment parameters are passed directly to the physics model and not to anything else on the CAN bus, this research has also separated it from the control systems of AMT and ABS. In this way, a solution to the problem with just 11 features was obtained from the GA with the expectation that it could at least find one on three ECUs, allowing the physics model and the environment variables to be kept on a fourth ECU. This is a real application of the GA, whereby engineers may choose to make their own decisions about separation of features before running the program.

1 run(s) of ECU/FEATURES PROBLEM with 11 features
Max iterations 2^20 = 1048576
percentage displayed 0.100000
Global iteration count = 1048575
Highest iteration number is 58
Average runtime after total number of runs = 7.9820000
Solution 1 is achieved 1 times or 100.00%:
solution is [0 7 7 5 5 5 7 7 0 7 0]
ECU 0: : Pedal : Clutch : AMT
ECU 5: : Wheels : Callipers : Throttle
ECU 7: : ABS : IMU : Gear Stick : Engine : Gear Box
DONE
THIS WAS THE NON-EXHAUSTIVE RANDOM GA MODEL
Reminder: CLOCKS_PER_SEC = 1000
This was run on the desktop machine
Main is ending. Programme finished!

Table 5-16 : Results of GA for ABS/AMT with 11 nodes

ECU/Feat	0	1	2	3	4	5	6	7	8	9	10
0	x								x		x
5				x	x	x					
7		x	x				x	x		x	

In the Arduino HILs execution, the environment variables for road/wheel friction are tested manually by the driver on start-up and AMT successfully manages gear changes and throttle input to increase the road speed of the vehicle before brake pedal input is interpreted by the ABS controller to increase the brake demand at the callipers and decrease the wheel speeds and vehicle speed. The result of a single run of the AMT/ABS HILs is shown in figure 5-7 with speeds for each wheel (FL, FR, BL, BR) and the vehicle's road speed as the gear selection and throttle application increase the speed up to 90m/s at event 14, before a momentary lapse on the throttle control causes a reduction and then increase between 77 and

90m/s at events 15 and 16. The brakes are applied after event 16 and the deceleration of the vehicle can be seen as it slows to standstill due to the application of the brakes.

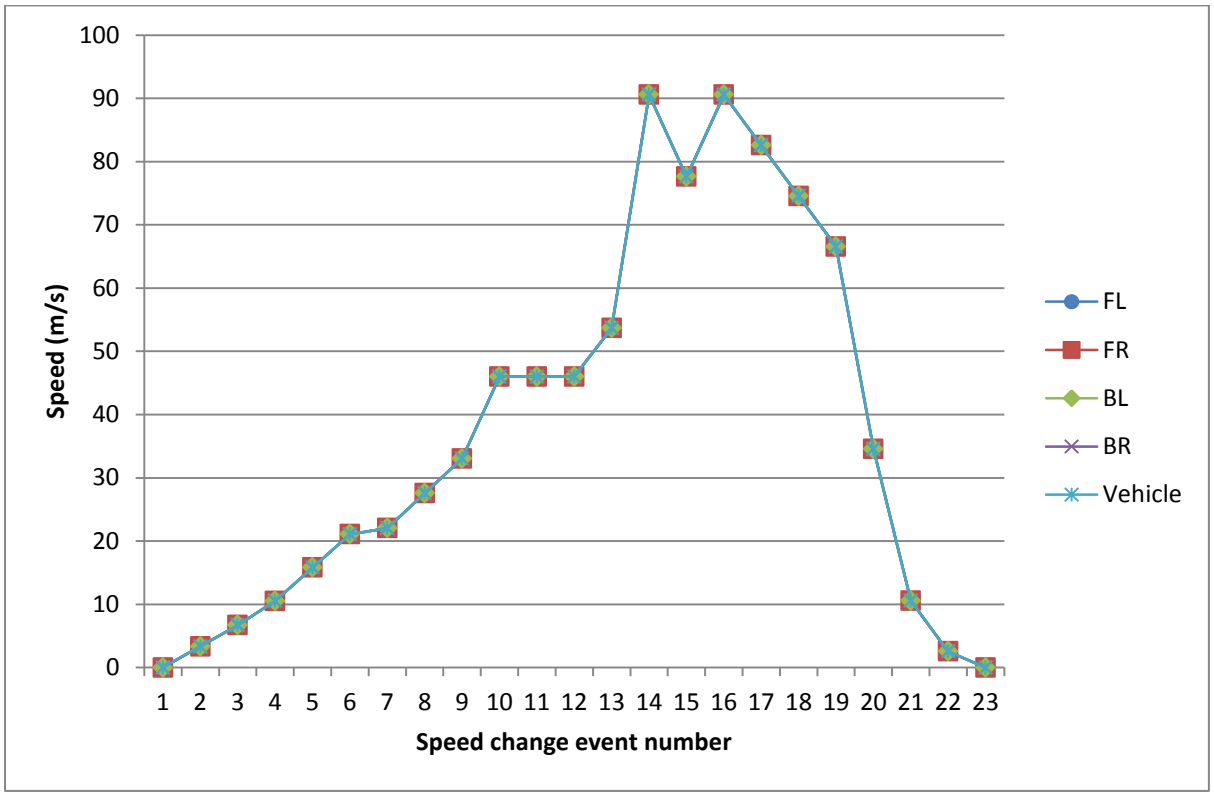


Figure 5-7 : Integrated AMT and ABS showing a single acceleration/braking event

In the appendices, textual output from each of the four ECUs shows the sequence of HMI input and controller responses with CAN messages and calculated vehicle speeds from simulated throttle changes and brake pedal pressures. In the 'Calliper Throttle and Wheels' program, in Appendix J, wheel speeds increase from zero to 90.57 m/s in steps of approximately 3 m/s followed by a braking event with deceleration of 8m/s/s down to zero. The physic model, Appendix K, clearly shows the result of testing the environment variables from 8 down to 1 and back up to 8 again. CAN messages from the gear stick (60) the AMT controller (80) and the gear number from the gear box (180) are shown to have been received before the AMT and physics models make use of these and attempts by the physics model to calculate the new road speed and wheel speeds are shown to have succeeded. In Appendix L, the brake pedal program that shares the ECU with the AMT and clutch software is seen to send brake pedal information via CAN and to manage upshifts and downshifts requested by the gear stick. The 'Engine gearbox gearstick IMU' node (Appendix M) receives messages from the physics model to update the road speed due to both braking events and acceleration events generated by changes in engine revs from the throttle and selection of different gears.

5.6 Conclusions

An end to end model of an integrated ABS/AMT system, that validated the design process, was achieved on Arduino experimental prototyping boards. The individual features of the

system were supported on individual ECUs with bespoke software for the control algorithms as an outcome of this research and third party software being acquired for low level drivers and communication protocols. A reduction in the number of ECUs needed to support the features was obtained from executing the GA and HMIs were connected to the μCs to achieve a HIL simulation using four ECUs and as many logarithmic potentiometers to simulate the brake pedal, throttle, gear lever and environment variables for friction.

More features and more variables in the ECU attributes – for example, the GA could process features with greater requirements such as temperature ranges, vibrational limitations, electromagnetic interference and distance/proximity to and from other components of the vehicle and attributes of the ECUs that would be able to support these.

6. Conclusions

The aim of this research was to develop a process that would allow the rapid prototyping of distributed real-time embedded system models that can be executed for verification purposes. This was realised in software/hardware on desktop PCs and on networks of microcontrollers physically connected by wired communications protocols, specifically CAN.

Models were designed by use of UML diagrams, generation of code that produced stubs for 'C' programs, system models of ABS and AMT, an integrated model that simulated the functions of control systems in a vehicle and the simulation of the vehicle dynamics by a physics model that was programmed to calculate values of road speed and wheel speeds used by the ABS controller in order to modulate brake demand in an emergency such as skidding on a slippery road surface.

The proposed design process used systems modelling diagrams in UML to create documentation that could be shared between engineers and project managers for the purposes of communication of ideas and for generating some of the code that went into the individual μCs of the distributed system. Because of constraints on finances, access to licences for some of the proprietary professional UML software such as 'IBM Rational Rhapsody' and 'LieberLieber Embedded Engineer' was not possible. Both of these promised to automatically generate fully functioning embedded code on microcontrollers from UML diagrams (although no end to end example was found of such) and this research relied on an open source freeware product, 'BridgePoint xtUML' which was capable of generating executable code on a desktop Windows PC for the purposes of testing and simulating a system of distributed electrical and electronic components. Some source code generated by this software package was manually tailored to compile and execute on microcontrollers although the benefits were offset against time taken to learn a new proprietary action language that then translated into 'C'.

An initial hardware/software architecture, based on the number of features in an ABS/AMT system, was modelled and documented in xtUML. This allowed changes to be made to the design with reduced effort while the class descriptions and methods of each element of the system could be amended if necessary before the process of generating stubs of 'C' source code took place.

A simulation of a microwave oven, which was included gratis as part of the xtUML download, provided an end to end example of a system design from documentation to a simulation on a desktop PC. Specifications and requirements for the design were missing but implied by the detail in the systems diagrams and, whilst there was no suggestion that a microwave oven had actually been designed, the simulation demonstrated the various states of the oven over time along with the sequence of events in a typical use case. Timings for cooking events ranging from opening and closing the door of the microwave oven to initiating the magnetron tube and setting different power levels for the cooking process were demonstrated by the example simulation when compiled and executed. The usefulness of the

automatically generated code was limited to simply running down a clock timer and displaying the names of some events such as “Door open”, “Door closed”, “Tube on”, “High power” etc.. This did not cause an actual microwave oven to do anything and there was little in the way of anything that would code the transfer of data for communications purposes between individual ECUs, should that be the implementation of the system.

Whilst this example could be altered to suit bespoke systems, generating source code in ‘C’ that would be compiled into an executable simulation, it could not be used to generate the executable code that would run on microcontrollers for the purposes of control and communication across nodes of a wired distributed embedded system of ECUs. UML generally and xtUML were disappointing in this respect but did have limited usefulness in terms of documentation for communication of ideas.

Attempting to use this tool to generate executable code for the ABS/AMT case studies was therefore fraught with frustration and disappointment. Completely coding the microcontrollers of the ABS/AMT systems automatically from diagrams and action language was not possible and instead xtUML was used only to generate minimal stubs of ‘C’ source code that could define the different classes and structures of entities like the brake pedal, wheels, callipers, IMU and ABS controller. Even with action language coded into the class diagrams as shown in Figure 4-3, further manual coding of the individual ‘struct’ types of those entities was required in order to fill the classes and methods with meaningful source code or with any source code in some instances. For example, the code generated from the ‘ABS_Control’ class contained empty stubs of the methods defined in Figure 4-3 and nothing else of any use in programming a microcontroller board to perform the functions of this vehicle system.

The use of ‘action language’ in xtUML did mean that much of the auto-generated source code was simply a translation of code in the proprietary language that could have easily been written in ‘C’ or some other existing language to avoid the necessity for learning another set of coding vocabulary and syntax. This research has not analysed the benefits or otherwise of taking time to learn the new action language compared with any time saving that would have occurred from the UML package being able to compile ‘C’ directly. In conclusion, whilst UML and in particular xtUML can be used to automatically generate source code in the ‘C’ language, the limited amount of useful code that can be created automatically for embedded systems of ECUs meant that, at present and until more automation is possible, xtUML is not currently a substitute for the manual coding skills of a software engineer, but it does have value in the design and documentation of the system.

The process of designing a tool, that would automatically solve the problem of how to embed code for each individual feature into fewer ECUs than there were features in the system, was very successful. Both a GA and a bin packing tool were developed, in the research process, such that the number of ECUs used to implement the vehicle control systems (ABS and AMT) was fewer than the number of features and such that code for more than one feature could be executed on the same ECU. For known sizes of executable code, including proprietary third

party libraries and low level drivers, an optimisation algorithm designed in this research was able to allocate hardware/software resources to each feature, sharing multiple features on at least one ECU, thereby reducing the total number of ECUs required to fewer than the overall number of features.

Because of the limitations of code generation by the xtUML package, much of the code that did form the control programs for each feature of the vehicle came from the author's manual intervention. Very little code was obtained from automation or from third parties as would be the case in industry where a choice could be made to offset the convenience of automatic code generation against the control and the ability to understand the complexity of executables designed manually by engineers in each of the vehicle domains. In real-world examples of features that are not designed by the vehicle's OEM and are provided by tier 1 or tier 2 companies, the software to control those features would not be written by the OEM except for some very limited ability to configure or calibrate some of the fixed values for things like wheel/tire sizes, for example, that would be needed for the wheel speed sensors to correctly calculate road speeds. This restriction on any hardware/software architectures or examples of industrial code being made available by vehicle OEMs severely hampered progress and affected the ability to accurately compare architectures to accurately measure the success and the specific benefits to industry of the optimisation tool. General savings made by the design of the optimising tool in this research are discussed later in these conclusions.

Having manually written code segments that would execute the control of each feature, this research created a design tool that delivered solutions to the problem of assigning and grouping the code on to fewer ECUs. This allowed the HILs on Arduinos to be designed from known values for memory requirements of control algorithms that were output from this research. The allocation tool delivered solutions that were implemented on Arduino μCs connected via CAN and these were demonstrated to function correctly and in a timely fashion. Real time output results of execution from the deployed network of Arduino microcontrollers simulating ABS/AMT are shown in appendices J to M.

The physics model, as a simulation of the vehicle dynamics, resided on a node of the Arduino HILs and communicated with the ABS and AMT via CAN. Because of this, the timings and the communication of the vehicle features, the controllers and the other things were affected by the communications and the processing time of the physics model, which ate into the bandwidth and took processing time away from the normal function of the ABS controller and the features of the ABS. This meant that some of the timings appeared to have too much latency and would otherwise have missed system safety critical deadlines. Subtracting the physics model's timings from the CAN bus communication times restored the simulation's results to a position where they could be considered safe. Generally, the aim of this research was realised successfully with an end to end example of a system designed with UML and bespoke optimisation tools.

Considering the objectives in turn and how they were accomplished, they were…

Objective to identify practices, methods and architectures in design and implementation - current practices, revealed by the literature search, indicated that there were few if any realistic ways of using systems modelling diagrams to produce working code for implementation on μCs for real time embedded systems, although a limited amount of source code that amounted to stubs of 'C' code was possible.

Objective to implement models to represent vehicle electronics, ECUs, communications buses and constraints – models were created in xtUML and in hardware/software on Arduino experimental prototyping μC boards.

Objective to obtain and develop suitable application code and software patterns that implement the features and low-level device drivers – whilst it was anticipated that these could be sourced from third parties, this did not happen as planned and, consequently, all application code to implement the features was written by the author of this research thesis.

Objective to automate the process to generate suitable executable models – various executable models were envisaged for this research from UML diagrams (that would produce code to execute on a desktop PC to simulate a hardware/software system such as the xtUML microwave oven that was adapted to become an ABS model) to a solving tool that assigned ECUs to features and a hardware/software simulation of distributed networks of vehicle features, realised on Arduino μCs.

Develop a tool for the allocation of features to the distributed system – two heuristic methods, a genetic algorithm and a bin-packing process, were coded into a tool that took attributes of available ECUs and requirements of vehicle features as input and produced, as output, an optimised software/hardware architecture and allocation of ECUs to multiple shared features.

Objective to develop a suitable approach for the testing of the executable model – the ECU/features allocation tool was tested with data for different scenarios from a small system of three features to up to 100 features using different sizes of ECU with varying memory capacities and processor speed attributes. ABS HILs was tested using Arduino μCs that took the code written for cross-platform implementation (desktop/ECUs) and brought separate, modular, code together on the same ECUs to reduce the number of ECUs used.

The HILs simulation, deployed on a network of Arduino microcontroller boards and connected with wired CAN communications protocol, demonstrated the success of the ECU/features allocation tool and was both verification and validation of the design process and the control algorithms. The hardware/software simulation successfully demonstrated the sharing of multiple features' control software and the optimisation of the number of ECUs, required in supporting the features. Human machine interfaces, such as switches for gear selection and pedals for accelerator and brakes were successfully simulated by logarithmic potentiometers and software to interpret and filter the user requests that would normally be achieved by pedals, levers, paddles and switches operated by the hands and feet.

In the 10 item knapsack problem, there are $2^{10} - 1$ possible solutions, not including leaving the knapsack empty. All solutions are possible. That is any combination of any number of items may be placed in the sack but each solution must be tested against the constraints on weight limits and the fitness of the maximum value. Since it can't be determined outside of the algorithm how many items need to be left out of the knapsack (if any) there are no shortcuts to predicting the optimal solution other than testing alternative candidate solutions by creation of chromosomes by the random generation and mutation of offspring.

In the ECU/features problem, random changes are made to the offspring by crossover and mutation. These values have been arbitrarily chosen and some investigation of the effects of changing these is recommended for future research. The non-continuous, discrete, nature of the ECU problem meant that it could not be solved by derivative-based search algorithms that use the gradient of a function or curve to determine the direction of the search towards local and global maxima or minima. For the continuous model, heuristic methods move up or down a slope in a landscape of mountains and valleys but, for the discrete model, information about the gradient and the value of neighbouring solutions is missing from any single candidate solution. Further research into the multidimensional neighbours, which was beyond the scope of this research, is suggested.

Because of the random nature of the changes to each subsequent generation in the GA, caused by crossover and mutation, the algorithm for this GA is non-deterministic and, other than by chance, the same result is not expected in subsequent runs with the same starting criteria. Also, because of the limited number of iterations, deliberately specified so as to reduce the total execution time by examining only a subset of the search space, not every possible solution will be examined and it cannot be predicted which will and which won't. This means that there is no guarantee of finding the global maximum or the optimal solution unless the GA were allowed to run with sufficient generations of new populations and with sufficient history of previously generated genes as to prevent any DNA being generated twice. A recommendation for further research into GAs for this problem is to prevent the generation and evaluation of DNA that has either already been generated or that has been assigned a fitness score that is lower than the current best score. By this method, execution of the GA could theoretically reach the optimal solution by generating sufficient DNA as to cover the entire search space, thereby performing an exhaustive search as a result of stopping criteria that allow sufficient time and sufficient unique DNA to generate every possible gene representing every possible solution.

Genetic algorithms use a sequence of operations and steps that alter the DNA in order to improve upon solutions by evolving over time. This does not necessarily have to be a computerised or automated process but the digital, often binary nature of the chromosomes in a GA means that it can be automated, computerised and parallelised to run very quickly on desktop computers and/or supercomputers, examining many more candidate solutions than a human could do in the same amount of time either manually or by inspection. The generation of new child chromosomes could be performed manually by a human in a random or arbitrary fashion and the user's input analysed by programmed executable code. This could be done in

real time with a graphical user interface (GUI) such as an input screen pro forma or from prepared data saved in files to be read in by the program at execution time. It could also be manually edited and hard coded in to the source files as declarations of data structures or arrays that would be compiled in to the executable code.

Subsequent runs of the executable models would require a re-write of the data inside the body of the source files but a possible advantage of this would be that the human is unlikely to enter the exact same chromosome twice unlike the GA which has no specific guards against this happening, a chromosome that was generated much earlier in the run being produced again more than once or twice later on.

Considering whether the GA is any better at producing useful chromosomes if it is only randomly changing some elements of a 0-1 binary array/vector, the answer is, unfortunately, "No" unless some other checks are made at each step of the generation of new offspring. For example, if the chromosome length is not too large (possibly in the range of 100x100 or 120x120) a trawl can be made through each element of the chromosome to check if it has been generated before. This could be by storing a decimal representation of each chromosome in a lookup table or by 'anding' the current chromosome with the known ones that have been created in this execution and stored in a history list or file. A recommendation that this be part of future research is made here.

Another useful method, not employed in this research but worthy of consideration for future work, is to use the bin packing algorithm to determine a starting point for the GA followed by a run of the GA with the solution offered by the bin packing as its own starting point. The combination of the two executables would deliver a chromosome that could be used to produce a better solution than the diagonal solution of 1 ECU for every feature. From this starting point, the GA would never suggest the use of more ECUs than the BP had delivered but information about inferior chromosomes that had either been discarded or already generated would be missing and would need to be generated from scratch.

Whilst the BP finds solutions much faster than the GA for the same size of problem, there are scenarios in which the BP cannot ever find the optimal solution where the GA could. For example, to demonstrate this, in the simplest case of just three ECUs and three features, consider bins with capacities of 4, 3 and 2 units respectively attempting to be allocated to items of size 3, 2 and 2 units. Figure 6-1 shows that the BP algorithm will begin with all of the bins and all of the items in capacity and size orders respectively, meaning that the first operation will assign bin 1 to item 1 leaving capacity of only one unit in bin 1.

Now the algorithm will assign bin 2 to item 2, because it won't fit in the single unit remaining in bin 1. Bin two will be left with just one unit remaining. The last item will only fit in bin 3 and the algorithm stops with three bins being used to solve the problem.

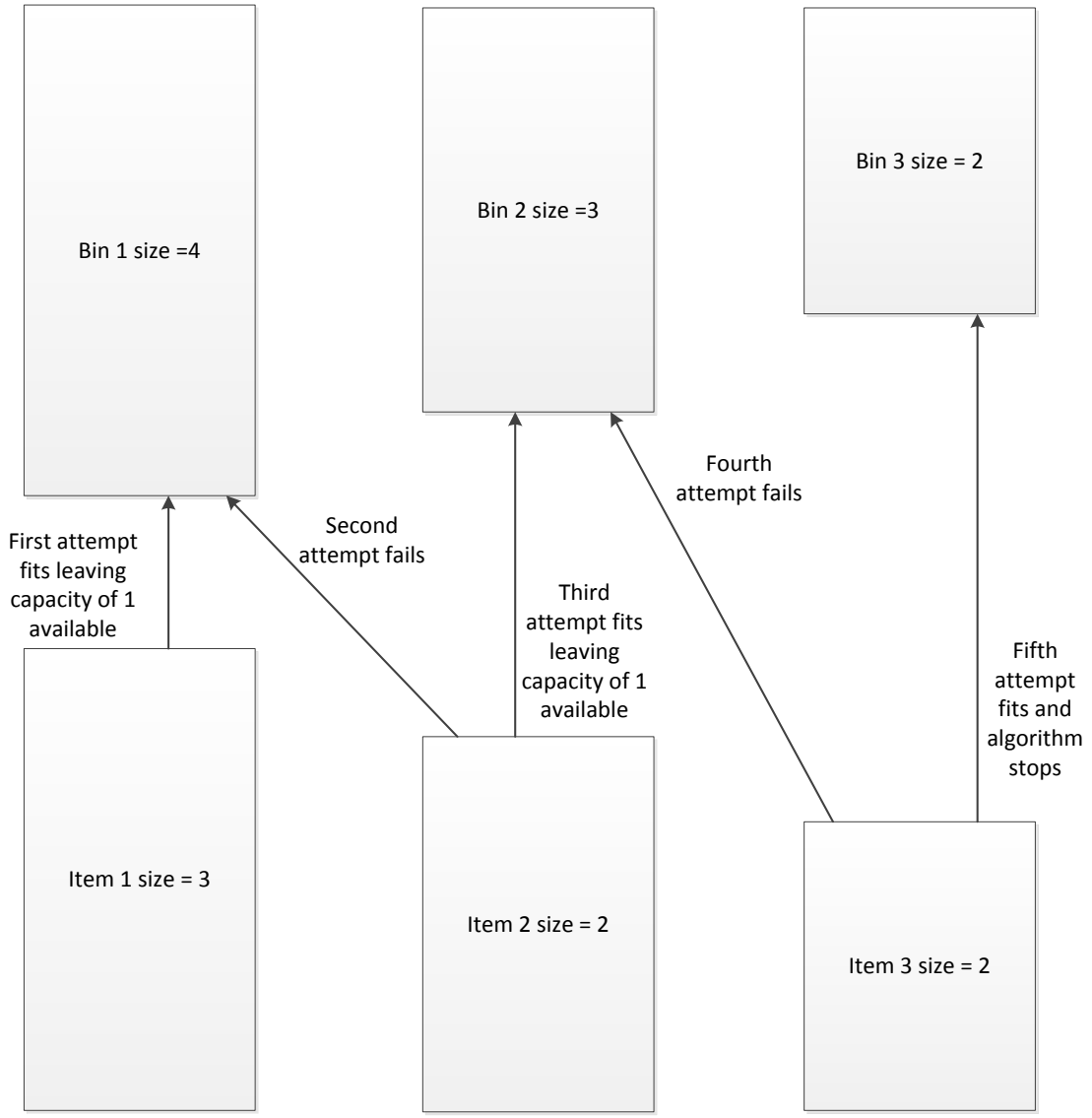


Figure 6-1 : Simple sub optimal bin packing scenario

Clearly, the optimal solution for using the fewest bins would have been to assign both items, 2 and 3, to bin 1 and item 1 to bin 3. Now, consider the same problem being solved by the GA. The exhaustive model will examine the chromosome [011 100 000] and find that there is a solution in which only two ECUs are used – two items of size 2 will fit into the bin with capacity 4 and the remaining item fits into the bin with capacity 3. This solution could feasibly be found by the GA but the probability that the BP would find it is zero, because of the order in which the BP attempts to fit the items to the bins.

An interesting line of research for the future is the multi-dimensional geometries that identify the feasible chromosomes of the GA for the mapping problem using an n x n array and the connection this may have to the adjacency matrix of a directed graph. This could possibly be a way to design a tool that would replace the GA by use of geometric algebras and executable code that would traverse a graph of the kind that could be drawn from the n x n matrices.

There is potential for research into the networks of communications and power cables that make the CAN and electrical-power looms of the vehicle. Routing of the cables and positioning of the ECUs at strategic points around the vehicle could reduce the length of wiring and therefore the overall mass of the vehicle to increase performance. The industrial sponsor of this research uses a proprietary tool to design and optimise the routing of power cables but the protection of their intellectual property prevented this from being made available to investigate or analyse its utility and research into similar tools is a topic for the future. This research has presumed the use of physical wiring for the CAN communications bus and whilst the author acknowledges the use of other communications protocols in the automotive industry, no other communication protocol was investigated thoroughly and simulated in the way that CAN was.

It is also recognised that there is a move towards wireless i/o in some distributed systems of μCs. Whilst this would allow for reductions in mass of wiring, there is a security implication with respect to hacking the vehicle at vulnerable points in the wireless communication that have not been fully researched. This research did not attempt to model or simulate wireless i/o communications and this could be tackled in future research into similar design methods for systems of ECUs. The automated process that assigns ECUs and features of the vehicle does not include provision for solving problems of electrical wiring networks or identification of best communications protocols to be implemented. This could form the basis of further work towards a fully automated system of design of distributed embedded systems, network architecture and communications.

The ECU/features allocation solving tool was not designed to make decisions about which features should be grouped together on any single ECU. That decision is left to the user and engineers. Using the tool in carefully managed stages did allow for small subsystems of both the ABS and the AMT to be considered in isolation. When the entire ABS system was analysed by the tool, candidate solutions arose that made suggestions about which features could be shared, without consideration of weather it made sense, for example, to have the software that controlled the brake pedal on the same ECU as the software for a brake calliper. These can be thought of as grammatically syntactic and semantic solutions, in the same way that a sentence can be constructed according to grammatical rules but might contain words or word order that does not make sense semantically. Solutions that placed the pedal and the ABS controller on the same ECU did make more sense and this solution did show up but could also be promoted by careful placing of features into the ECU/features allocation solving tool, rather than placing 100 features and ECUs into the data for a single execution and expecting features to be allocated in a sensible way to neatly grouped ECUs, although this should not be ruled out of future research. Currently, the way in which the vehicle is divided and classified into domains and systems of smaller features and subsystems is a decision made by humans. This could be researched in future to create tools that make these decisions as part of the automated design process.

In this research, only a few variables were considered as attributes of the ECUs or as requirements of the vehicle features. A simple extension to this research would be to include

more variables into the GA and BP tools so that other parameters such as vibration sensitivity, temperature range, proximity to electromagnetic sources, mass etc., could be included in the fitness score when assigning an appropriate ECU to a feature. Whilst this could produce a solution based on more complex relations of variables and attributes of the features and ECUs and whilst this would be desirable for vehicles systems, other systems that could benefit from the design process, for example the medical care systems mentioned earlier, might not benefit from such complex design considerations.

Savings resulting from the design process proposed in this research arise from various areas. The novel use of the GA combined with predetermined constraints to dismiss infeasible solutions, based on geometries arising from the dimensionality of the number of ECUs and features, reduces the computation time for the GA compared with an exhaustive search or with a GA that does not filter out infeasible solutions before assigning a fitness score. The saving in execution time for this GA is by a factor of at least $0.5\left(2^{n^2}.n-n\right)$ for the resource allocation where *n* is the number of vehicle features.

Although difficult to quantify, it is argued that the time to market must be reduced by the inline reduction in execution time of the GA compared with the exhaustive search. For systems of up to 100 features this saving in execution time is actually immeasurable because an optimisation based on this many ECUs would take longer than the age of the universe to search exhaustively but a reasonable estimate is that this could reduce the overall time to market by weeks or even months. Other savings include overall cost of components where the number of ECUs is reduced, in the specific ABS/AMT case study from 13 ECUs to 3, giving a total cost saving for this example alone of over 75% on microcontrollers.

In conclusion, the research shows a viable design process that can generate code automatically, reduce the size of the search space for candidate solutions to problems of ECU allocation and test hardware/software architectures of distributed embedded systems and networks of wired communications protocols. The process reduces the number of ECUs in vehicles, the amount of wiring, mass of the vehicle, execution times of search algorithms and time to market, increasing performance and profit.

7. References

[1] J. Mauss and M. Simons, “Chip simulation of automotive ECUs,” in *Symposeum Steuerungssysteme fur automobile Antriebe*, 2012, no. September 2012.

[2] M. J. Pont, *Patterns for time-triggered embedded systems*. London: Addison-Wesley, 2014.

[3] H. Kashif, M. Mostafa, H. Shokry, and S. Hammad, “Model-based embedded software development flow,” *2009 4th Int. Des. Test Work.*, pp. 1–4, 2009.

[4] A. Rajnak and A. Kumar, “Computer-aided architecture design & optimized implementation of distributed automotive EE systems,” *Proc. - Des. Autom. Conf.*, pp. 556–561, 2007.

[5] O. Sander, T. Sandmann, D. V Vu, S. Baehr, F. Bapp, J. Becker, H. U. Michel, D. Kaule, D. Adam, E. Lubbers, J. Hairbucher, a Richter, C. Herber, and a Herkersdorf, “Hardware virtualization support for shared resources in mixed-criticality multicore systems,” *Des. Autom. Test Eur. Conf. Exhib. (DATE), 2014*, pp. 1–6, 2014.

[6] M. Becker, D. Dasari, V. Nelis, M. Behnam, L. M. Pinho, and T. Nolte, “Investigation on AUTOSAR-Compliant Solutions for Many-Core Architectures,” *2015 Euromicro Conf. Digit. Syst. Des.*, pp. 95–103, 2015.

[7] W. Won, J. Son, G. Park, D. Kum, and S. Lee, “Design and Implementation Procedure of the AUTOSAR I / O Driver Cluster,” *Design*, pp. 5618–5623, 2009.

[8] K. Senthilkumar and R. Ramadoss, “Designing multicore ECU architecture in vehicle networks using AUTOSAR,” *3rd Int. Conf. Adv. Comput. ICoAC 2011*, pp. 270–275, 2011.

[9] M. Hillenbrand and K. D. Müller-Glaser, “An Approach to Supply Simulations of the Functional Environment of ECUs for Hardware-in-the-Loop Test Systems Based on EE-architectures Conform to AUTOSAR,” *2009 IEEE/IFIP Int. Symp. Rapid Syst. Prototyp.*, pp. 188–195, 2009.

[10] D. Goswami, R. Schneider, A. Masrur, M. Lukasiewycz, S. Chakraborty, H. Voit, and A. Annaswamy, “Challenges in automotive cyber-physical systems design,” *Proc. - 2012 Int. Conf. Embed. Comput. Syst. Archit. Model. Simulation, IC-SAMOS 2012*, pp. 346–354, 2012.

[11] D. Schreiner and K. M. Göschka, “A component model for the AUTOSAR virtual function bus,” *Proc. - Int. Comput. Softw. Appl. Conf.*, vol. 2, no. i, pp. 635–641, 2007.

[12] P. Pop, P. Eles, Z. Peng, and T. Pop, “Analysis and optimization of distributed real-time embedded systems,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 3, pp. 593–625, 2006.

[13] R. hvanth, D. Valli, and K. Ganesan, “Design of an In-Vehicle Network (Using LIN, CAN and FlexRay), Gateway and its Diagnostics Using Vector CANoe,” *Am. J. Signal Process.*, vol. 1, no. 2, pp. 40–45, 2012.

[14] C. P. Quigley, R. Mcmurran, and R. P. Jones, “An Investigation into Cost Modelling for Design of Distributed Automotive Electrical Architectures,” *2007 3rd Inst. Eng. Technol. Conf. Automot. Electron.*, pp. 1–9, 2007.

[15] W. Voss, *A Comprehensible Guide to Controller Area Network*, 2nd ed. Copperhill Technolgies Corporation, 2005.

[16] C. Wey, C. Hsu, K. Chang, and P. Jui, “Enhancement of Controller Area Network (CAN) Bus Arbitration Mechanism,” in *Connected Vehicles and Expo (ICCVE), 2013 International Conference on*, 2013, pp. 898–902.

[17] W. Voss, *Controller Area Network Prototyping with Arduino*. Grrenfield, MA: Copperhill Technolgies Corporation, 2014.

[18] A. Forsberg and J. Hedberg, “Comparison of FlexRay and CAN-bus for Real-Time Communication.”

[19] S. C. Talbot and S. Ren, “Comparision of FieldBus Systems CAN, TTCAN, FlexRay and LIN in Passenger Vehicles,” pp. 26–31, 2009.

[20] “AUTOSAR.” [Online]. Available: http://www.autosar.org/about/technical-overview/virtual-functional-bus/. [Accessed: 20-Nov-2015].

[21] M. Laifenfeld and T. Philosof, “Wireless Controller Area Network For In-Vehicle Communication,” *2014 IEEE 28-th Conv. Electr. Electron. Eng. Isr.*, no. 1, pp. 1–5, 2014.

[22] P. Guo, Y. Li, P. Li, S. Liu, and D. Sun, “A UML Model to Simulink Model Transformation Method in the Design of Embedded Software,” *2014 Tenth Int. Conf. Comput. Intell. Secur.*, pp. 583–587, 2014.

[23] T. G. Moreira, M. a. Wehrmeister, C. E. Pereira, J. F. Pétin, and E. Levrat, “Automatic code generation for embedded systems: From UML specifications to VHDL code,” *IEEE Int. Conf. Ind. Informatics*, pp. 1085–1090, 2010.

[24] M. Ibrahim and R. Ahmad, “Class diagram extraction from textual requirements using natural language processing (NLP) techniques,” *2nd Int. Conf. Comput. Res. Dev. ICCRD 2010*, pp. 200–204, 2010.

[25] H. Burden, R. Heldal, and T. Siljamäki, “Executable and translatable UML - How difficult can it be?,” *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, pp. 114–121, 2011.

[26] P. H. Hughes and J. S. Løvstad, “A generic model for quantifiable software deployment,” *2nd Int. Conf. Softw. Eng. Adv. - ICSEA 2007*, no. Icsea, 2007.

[27] J. Miller and J. Mukerji, “MDA Guide Version 1.0. 1,” *Object Manag. Gr.*, vol. 234, no. May, p. 51, 2003.

[28] A. Costa, V. Pazzini, L. Foss, S. A. D. C. Cavalheiro, L. B. De Brisolara, and F. R. Wagner, “Automatic Translation from UML to Simulink CAAM Using Graph Grammars,” *2013 2nd Work. Theor. Comput. Sci.*, pp. 59–66, 2013.

[29] P. Romaniuk, “Translator of hierarchical state machine from UML statechart to the event processor pattern,” *Proc. 14th Int. Conf. "Mixed Des. Integr. Circuits Syst. Mix. 2007*, pp. 684–687, 2007.

[30] S. Cherif, I. R. Quadri, S. Meftali, and J. L. Dekeyser, “Modeling reconfigurable systems-on-chips with UML MARTE profile: An exploratory analysis,” *Proc. - 13th Euromicro Conf. Digit. Syst. Des. Archit. Methods Tools, DSD 2010*, pp. 706–713, 2010.

[31] S. Demathieu, F. Thomas, C. Andre, S. Gerard, and F. Terrier, “First Experiments Using the UML Profile for MARTE,” *2008 11th IEEE Int. Symp. Object Component-Oriented Real-Time Distrib. Comput.*, no. 1, pp. 50–57, 2008.

[32] R. Chen, M. Sgroi, L. Lavagno, a. Sangiovanni-Vincentelli, and J. Rabaey,

"Embedded system design using UML and platforms," *Syst. Specif. Des. Lang.*, pp. 119–128, 2003.

[33] G. Bazydlo, M. Adamski, M. Wegrzyn, and A. Rosado Munoz, "From UML specification into FPGA implementation," *Adv. Electr. Electron. Eng.*, vol. 12, no. 5, pp. 452–458, 2014.

[34] J. Park, B. Wang, J. Jeon, and S.-H. Hwang, "Hardware in-the-loop simulation for ABS using 32-bit embedded system," *Control. Autom. Syst. (ICCAS), 2011 11th Int. Conf.*, pp. 575–580, 2011.

[35] D. Xue, X. Yin, and L. Li, "Software architecture for the ECU of automated manual transmission," *Softw. Eng. Data Min. (SEDM), 2010 2nd Int. Conf.*, pp. 63–68, 2010.

[36] D. K. Hammer and L. R. Welch, "Special Issue On Object-oriented Real-time Systems: Guest Editor's Introduction," *Jpdc*, vol. 36, no. 36, pp. 1–3, 1996.

[37] P. Pesonen and V. Seppanen, "Object-based design of embedded software using real-time operatingsystems," *Proc. Sixth Euromicro Work. Real-Time Syst.*, pp. 194–198, 1994.

[38] A. S. Fukunaga and R. E. Korf, "bin completion algorithms for multicontainer packing and covering problems," *J. Artifical Intell. Res.*, vol. 28, no. 1, 2007.

[39] R. Bar-yehuda, A. R. I. Freund, and J. S. Naor, "A Unified Approach to Approximating Resource Allocation and Scheduling," vol. 48, no. 5, pp. 1069–1090, 2001.

[40] M. Fleischer, Rudolph; Wahl, "On-line scheduling revisisted," in *Journal of Scheduling*, vol. 3, no. 6, 2000, pp. 343–353.

[41] R. L. Graham, "Bounds for certain multiprocessing anomolies.pdf," *Bell Syst. Tech. J.*, 1966.

[42] R. R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek, "Practical Solutions for QoS-based Resource Allocation Problems," *Proc. 19th IEEE Real-Time Syst. Symp. (Cat. No.98CB36279)*, pp. 296–306.

[43] T. Katoh, N; Ibaraki, "Resource allocation problems," in *Handbook of Combinatorial Optimization*, Springer, Boston, MA, 1998.

[44] R. Ramanathan and L. S. Ganesh, "Using AHP for resource allocation problems," *Eur. J. Oper. Res.*, vol. 2217, no. 93, pp. 410–417, 1995.

[45] J. L. Verdegay, R. R. Yager, and P. P. Bonissone, "On heuristics as a fundamental constituent of soft computing," *Fuzzy Sets Syst.*, vol. 159, no. 7, pp. 846–855, 2008.

[46] V. M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. Comput.*, vol. 37, no. 11, pp. 1384–1397, 1988.

[47] X. Xu, H. Yuan, M. Liptrott, and M. Trovati, "Two phase heuristic algorithm for the multiple-travelling salesman problem," *Soft Comput.*, vol. 22, no. 19, pp. 6567–6581, 2018.

[48] N. Gupta, "GENETIC ALGORITHMS : A PROBLEM SOLVING APPROACH What is Genetic Algorithm ?," no. 1975, pp. 940–944, 2012.

[49] J. H. Holland, "Genetic Algorithms - scientificamerican0792-66.pdf," *Sci. Am.*, no. July, 1992.

[50] J. Jain, N.K., Nangia, Uma; Jain, “Effect of Population and Bit Size on Optimisation of Function by Genetic Algorithm,” in *International Conference on Computing for Sustainable Global Development (INDIACom)*, 2016, pp. 189–194.

[51] P. A. Diaz-Gomez and D. Hougen, “Initial Population for Genetic Algorithms: A Metric Approach.,” *Proc. 2007 Int. Conf. Genet. Evol. Methods*, pp. 43–49, 2007.

[52] O. Roeva, “Influence of the Population Size on the Genetic Algorithm Performance in Case of Cultivation Process Modelling,” pp. 371–376, 2013.

[53] S. Gotshall and B. Rylander, “Optimal Population Size and the Genetic Algorithm.”

[54] K. K. Yit and P. Rajendran, “Investigation on Selection Schemes and Population Sizes for Genetic Algorithm in Unmanned Aerial Vehicle Path Planning,” pp. 6–10, 2015.

[55] A. Aleti, “Designing automotive embedded systems with adaptive genetic algorithms,” *Autom. Softw. Eng.*, vol. 22, no. 2, pp. 199–240, 2015.

[56] S. Kirkpatrick, “Optimization by Simulated Annealing,” vol. 220, no. January 1983, 2014.

[57] R. A. Zeineldin, “An Improved Simulated Annealing Approach for solving the Constrained Optimization Problems,” *2012 8th Int. Conf. Informatics Syst.*, pp. 27–31.

[58] T. El-Ghazali, *METAHEURISTICS*. Hoboken, New Jersey: John Wiley & Sons, Inc.,.

[59] S. Martello and P. Toth, *Knapsack Problems*. Chichester, UK: John Wiley & Sons, 1990.

[60] M. Yue, “A simple proof of the inequality FFD (L) ≤ 11/9 OPT (L) + 1, ∀L for the FFD bin-packing algorithm,” *Acta Math. Appl. Sin.*, vol. 7, no. 4, pp. 321–331, 1991.

[61] A. Alahmadi, A. Alnowiser, M. M. Zhu, D. Che, and P. Ghodous, “Enhanced first-fit decreasing algorithm for energy-aware job scheduling in cloud,” *Proc. - 2014 Int. Conf. Comput. Sci. Comput. Intell. CSCI 2014*, vol. 2, pp. 69–74, 2014.

[62] P. K. Yadav and N. L. Prajapati, “An Overview of Genetic Algorithm and Modeling,” vol. 2, no. 9, pp. 1–4, 2012.

[63] L. Austin and D. Morrey, “Recent advances in antilock braking systems and traction control systems,” *Proc. Inst. Mech. Eng. Part D J. Automob. Eng.*, vol. 214, no. 6, pp. 625–638, 2000.

[64] M. Watany, “Performance of a Road Vehicle with Hydraulic Brake Systems Using Slip Control Strategy,” *Am. J. Veh. Des.*, vol. 2, no. 1, pp. 7–18, 2014.

[65] A. K. Maurya and P. S. Bokare, “STUDY OF DECELERATION BEHAVIOUR OF DIFFERENT VEHICLE,” vol. 2, no. 3, pp. 253–270, 2012.

[66] N. Ding and S. Taheri, “A Modified Dugoff Tire Model for Combined-slip Forces,” *Tire Sci. Technol.*, vol. 38, no. 3, pp. 228–244, 2010.

[67] E. W. Dijkstra and E. W. Dijkstra, “On the Role of Scientific Thought,” *Sel. Writings Comput. A Pers. Perspect.*, pp. 60–66, 2012.

[68] N. Kudarauskas, “Analysis of emergency braking of a vehicle,” *Transport*, vol. 22, no. 3, pp. 154–159, 2007.

[69] H. B. Pacejka and E. Bakker, “The magic formula tyre model,” *Veh. Syst. Dyn.*, vol. 21, no. sup1, pp. 1–18, 1992.

[70] W. Wang, “Comparison of Aircraft Tire Wear with Initial Wheel Rotational Speed

Comparison of Aircraft Tire Wear with Initial Wheel Rotational Speed," *Int. J. Aviat. Aeronaut. Aerosp.*, vol. 2, no. 1, 2015.

[71] W. Voss, *A comprehensive guide to Controller Area Network*. Greenfield Massechussetts: Copperhill media, 2005.

[72] H. Katzan, "Operating systems architecture," p. 109, 2008.

[73] "Chevrolet Pressroom - United States - Spark EV." .

8. Appendices

Appendix A : Source code for algorithm to list number of solutions by row count

```
// Row_Counting.cpp : calculates the frequency of solutions with specific numbers of
rwos
//
#include <stdio.h>
#include <string.h>

#define NO_OF_ECUS 4
#define NO_OF_FEATURES NO_OF_ECUS
#define BASE_NO NO_OF_ECUS

int row_tally[NO_OF_ECUS]={0};
int chromosome[NO_OF_ECUS][NO_OF_FEATURES]={0};
char A[NO_OF_FEATURES]={'0'};
char Z[NO_OF_FEATURES]={'0'};

int rows(int ECUs)
{
	return(0);
}

void print_AZ()
{
	printf("\nString value A = ");
	for (int k=0;k<NO_OF_FEATURES;k++)
	{
		printf(" %i ",A[k]);
	}
	printf("\nEnding value Z = ");
	for (int k=0;k<NO_OF_FEATURES;k++)
	{
		printf(" %i ",Z[k]);
	}
}

void initialise_start_and_end_values()
{
	for (int k=0;k<NO_OF_FEATURES;k++)
	{
		A[k]=0; // make all elements of firt row = 0
		Z[k]=BASE_NO-1; // make all elements of last row one fewer than base
number
	}
	row_tally[0]=1;// set the first tally to record the [0 0 0] vector or top
features row of all ones - as this is not evaluated during the program execution.
}

void increment_row_one()
```

```
{
	int A_element=NO_OF_FEATURES-1;
	A[A_element]++;
	if (A[A_element]==BASE_NO)
	{
		do
		{
			A[A_element]=0;
			A[A_element-1]++;
			A_element--;
		}while ((A_element)!=0 && A[A_element]==BASE_NO);
	}
}

void print_chromosome_array()
{
	int row, col = 0;
	for(col=0;col<NO_OF_FEATURES;col++)
	{
		printf("\n[ ");
		for(row=0;row<NO_OF_ECUS;row++)
		{
			printf(" %i ",chromosome[row][col]);
		}
		printf("]\n");
	}
}

void test_row_count()
{
	int i,k=0;
	int row_count=0;
	for (i=0;i<NO_OF_ECUS;i++)
	{
		for(k=0;k<NO_OF_FEATURES;k++)
		{
			if(chromosome[k][i]==1)
			{
				row_count++;
				break;
			}
		}
	}
	row_tally[row_count-1]++;
}

void populate_chromosome()
{
	int A_count, row, col=0;
	for (row=0;row<NO_OF_ECUS;row++)
	{
		for(col=0;col<NO_OF_FEATURES;col++)
		{
```

```
                        chromosome[row][col]=0;
                }
        }
        for (A_count=0;A_count<NO_OF_FEATURES;A_count++)
        {
                chromosome[A_count][A[A_count]]=1;
        }
}


int main(int argc, char* argv[])
{
        int tally_element=0;
        initialise_start_and_end_values();
        do
        {
                increment_row_one();
                populate_chromosome();
                test_row_count();
        }while(strncmp(A,Z,BASE_NO)!=0);
        print_AZ();
        printf("\nRTV=[ ");
        for(tally_element=0;tally_element<NO_OF_ECUS;tally_element++)
        {
                printf(" %i ",row_tally[tally_element]);
        }
        printf("]\n");
        getchar();
        return 0;
}
```

Appendix B : Source code for the knapsack problem GA

```
// Knapsack.cpp : Defines the entry point for the console application.
//
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

int i,k=0;                          // global counter for use by main, init mutate and crossover
int r=0;                            // a random number
int no_of_runs=0;                   // number of times the main loop has been executed to determine end of execution
int actual_no_of_runs=0;            // number of times the main loop actually ran before stopping
int max_no_of_runs = 1550;          // the most times the loop should ever run
int min_no_of_runs = 100;           // the fewest number of times the main loop should ever run
int best_chromosome=0;              // the index of the best chromosome in any single pass of the main loop
int no_of_fits=1;                   // count from 1 to 4 of how many chromosomes have good fit at any one pass through the main loop

float c[4][10];     // c is for chromosome - arbtrarily four here - and any number of bits - 10 here
float v[10];        // v is for each item value of which there should be the same number as bits in any chromosome
float w[10];        // w is for each item weight available for the packing of which there should be the same number as bits in any
chromosome
float tw[4];        // tw is the total weight of any chromosome - indexed by array - in any single pass
float tv[4];        // tv is the total value of any chromosome - indexed by array - in any single pass

float weight_constraint;   // the upper allowable weight to be carried in the knapsack
float best_weight;         // the weight of the most valuable packed knapsack not exceeding the maximum allowed weight
float highest_value=0.0;   // the total value of the contents of the best packed knapsack considering both weight and value
float value_solution=0.0;  // updateable current value of the knapsack with the optimum packing
float weight_solution=0.0; // updateable current weight of the knapsack with the optimum packing

time_t t;                           // used to set the seed for the random number generator

void init_gene();                   // initialise the bits of each chromosome with a random number generator
void init_weights_and_values();     // initialise the bits of each weight array with constants
void select_genes();                // perform selection of the fittest chromosomes
void elite_select();                // perform elite selection to keep the best chromosomes for future generations
```

```
void swap_best();                // perform a swap with the first chromosome and the best of the curent population
void swap_second_best();         // perform a swap with the second chromosome and the best of teh rest after the first one
void random_select();            // perform selection of an arbitrary number of genes - let's say two in this case
void crossover();                // perform crossover to produce two new chromosomes
void mutate();                   // perform mutation by probablilty of 10%
void print_results();            // print the packing and values/weights of the optimally packed knapsack at end of run

int main()
{
      //printf("\nMaximise the console\n");getchar();
      init_gene();                                        // initialise the chromosomes by random 50/50 chance
      init_weights_and_values();                          // initialise weights and values from datafile or from hard-coded values

      while(no_of_runs < max_no_of_runs)                        // run the main loop for a maximum of max_no_runs
    {
          no_of_runs++;                                   // increment the current number of runs through the loop
          actual_no_of_runs++;                            // keep this in two places to find when the program stopped
          // for the four individual cases of chromosomes, find the best (highest) total value of the items in the sack
          for(k=0;k<4;k++)
          {
                tv[k]=0.0;                                // reset the total value  to zero before starting a loop
                tw[k]=0.0;                                // reset the total weight to zero before starting a loop
                for (i=0;i<10;i++)
                {
                      tv[k]=tv[k]+c[k][i]*v[i];           // cumulative sum of each bit multiplied by individual values of items
                      tw[k]=tw[k]+c[k][i]*w[i];           // cumulative sum of each bit multiplied by individual weight of items
                }
                //printf("\nThe total Monetary value of y[%i] is %f \n",k,tv[k]);
                //printf("\nThe total weight of y[%i] is %f \n\n",k,tw[k]);
                if (tv[k]>highest_value && tw[k]<=weight_constraint)     // only if the packing is underweight and greater than the
previous best value
                {
                      highest_value = tv[k];              // overwrite the highest value
                      best_chromosome = k;                // overwrite the best chromosome index
                      best_weight = tw[k];                // overwrite the weight for he best packed knapsack so far
                      if (highest_value>value_solution)   // only if the current high value is greater than the overall high value
                      {
                            value_solution=highest_value; // overwrite the final packing's value
                         weight_solution=best_weight;     // overwrite the final packing's weight
```

```
                }
            }
          }// finishing wrapping loop here
        //if(no_of_runs==1)                                   // put in to pause execution after first iteration
        //getchar();

        no_of_fits=1;                                         // reset the number of fits to equal just the first chromosome
        if(no_of_runs>min_no_of_runs)                         // only if the minimum number of runs has been reached
        {
            for(i=1;i<4;i++)
            {
                if(tv[i]==value_solution && tw[i]==weight_solution) // if any of the chromosomes matches the best result
                {
                    no_of_fits++;                                 // add one to the number of fits
                }
            }
        }

        if (no_of_fits>3)                                     // if an arbitrary number of fits has been found
        {
            no_of_runs=max_no_of_runs;                        // set the number of runs to equal the maximum allowed so as to
stop to program
        }
          if(no_of_runs!=max_no_of_runs)                             // only if the solution hasn't been reached
        {
               select_genes();                                // perform selection
                  crossover();                                // perform crossover
            mutate();                                         // perform mutation
          }
      }
      print_results();                                        // print out the results
      getchar();                                              // pause the program to view the results in the console
      return 0;                                               // return from main
}


void init_gene()
{
      srand((unsigned) time(&t));                             // set the seed for the random number generator
```

```
	for (k=0;k<4;k++)
	{
		for (i=0;i<10;i++)
		{
			c[k][i]=rand()%2;				// populate the bits of each chromosome with random 1s or 0s
		}
	}
	for (k=0;k<4;k++)
	{
		//printf("\nThe chromosome before crossover or mutation is [");
		for (i=0;i<10;i++)
		{
			//printf(" %i",(int)c[k][i]);		// print the initial genes to the console or file
		}
		//printf(" ]\n");
	}
	//printf("\n");
}


void init_weights_and_values()
{
	weight_constraint=25.0;
	best_weight=0.0;

	w[0]=5;
	w[1]=2;
	w[2]=3;
	w[3]=10;
	w[4]=7;
	w[5]=10;
	w[6]=5;
	w[7]=4;
	w[8]=18;
	w[9]=20;

	v[0]=10;
	v[1]=1;
	v[2]=15;
```

```
	v[3]=23;
	v[4]=9;
	v[5]=3;
	v[6]=21;
	v[7]=50;
	v[8]=22;
	v[9]=8;

	for (i=0;i<4;i++)
	{
		tw[i]=0.0;                                                                    //
initialise the total weights of each chromosome
		tv[i]=0.0;                                                                    //
initialise the total value of each chromosome
	}
}

void mutate()
{
    int nom;                                          // a variable for the number of mutations in any chromosome
	for (k=2;k<4;k++)
	{
		nom=0;
	 for (i=0;i<10;i++)
		{
			if((r=rand()%10)>8)                       // only 8 out of ten times according to modulo 10 division
			{
				nom++;                                                                // add one
to the number of mutations
		  //printf("\nThe random generator successfully produced : %i at y[%i][%i]\n",r,k,i);
				c[k][i]=(float)(((int)(c[k][i])+1)%2);   // flip any bit that has mutated from 1 to 0 or 0 to 1
		 }
		}
		//if (nom==0) //printf("\nThere was no mutation for chromosome %i \n",k);
	}

	//printf("\nThe chromosomes after mutation are\n");
	for (k=2;k<4;k++)
	{
```

```
            //printf("\ny[%i] =  [",k);
          for (i=0;i<10;i++)
            {
                  //printf(" %i",(int)c[k][i]);
            }
            //printf(" ]\n\n\n");
      }
}

void crossover()
{
      for(k=0;k<5;k++)                                          // for the first five bits
      {
            for(i=0;i<2;i++)                                    // and for the first two chromosomes
       {
           c[i+2][k]=c[i][k];                                   // copy the first five bits from c[0] to c[2] and from c[1] to
c[3]
       }
            for(i=3;i>1;i--)                                    // for the last two chromosomes
       {
            c[i][k+5]=c[3-i][k+5];                             // copy the last five bits from c[1] to c[2] and from c[0] to c[3]
       }
      }
      //printf("\nThe crossover has produced the following population\n");
      for (i=2;i<4;i++)
      {
            //printf("\nChromosome %i = [ ",i);
            for(k=0;k<10;k++)
            {
                  //printf("%i ",(int)c[i][k]);
            }
            //printf("]");
      }
      //printf("\n\n");
}
```

```
void select_genes()
{
	elite_select();                                                 // perform elite selection
	random_select();                                                // perform random selection
	//printf("\nThe chromosomes after elite selection are :\n");
	for (k=0;k<4;k++)
	{
		//printf("\n [");
		for (i=0;i<10;i++)
		{
			//printf(" %i",(int)c[k][i]);
		}
		//printf(" ]\n");
	}
	//printf("\n");
}

void random_select()
{


}


void elite_select()                                                 // pick the best two chromosomes and swap them with c[0] and c[1]
{
	swap_best();                                                    // swap the best chromosome with c[0]
	swap_second_best();                                             // swap the second best chromosome with c[1]
}

void swap_best()
{
	float temp_c[10];
	for (i=0;i<10;i++)
	{
		temp_c[i]=c[0][i];                                          // copies the previous best chromosome bit values into a
temporary holding array
		c[0][i]=c[best_chromosome][i];                              // copies the new best chromosome bit values into the first
chromosome position
```

```
            c[best_chromosome][i]=temp_c[i];              // copies the previous best chromosome bit values from temporary to
current best (before the swap)
      }
}

void swap_second_best()
{
      float temp_c[10];                                                                                    // set up a
temporary chromosome
      highest_value=0.0;                                                                                   // reset
the highest value to zer
      best_weight=0.0;                                                                                     // reset
the highest weight to zero
      best_chromosome=1;                                                                                   // the best
chromosome is the first one in the population
      for (k=1;k<4;k++)
      {
            tv[k]=0.0;                                                                                           //
reset the total value of each current chromosome to zero
            tw[k]=0.0;                                                                                           //
reset the total weight of each current chromosome to zero
            for (i=0;i<10;i++)
            {
                  tv[k]=tv[k]+c[k][i]*v[i];                                                          // cumulative sum
of values by bits
                  tw[k]=tw[k]+c[k][i]*w[i];                                                          // cumulative sum
of weights by bits
            }// end of for k = 0 to 10
            if (tv[k]>highest_value && tw[k]<=weight_constraint)  // only if underweight and total value exceeds the highest value
            {
                  //printf("\nBest Chromosome is overwritten here for k = %i\n",k);
                  best_chromosome=k;                                                                   // set the
current best chromosome index
                  highest_value=tv[k];                          // set the current best chromosome greatest value
                  best_weight=tw[k];                                                                   // set he
current best chromosome greatest weight
            }
      }// end of k from 1 to 4
      for (i=0;i<10;i++)
```

```
	{
		temp_c[i]=c[1][i];                                  // copies the second chromosome values into a temporary holding
array
		c[1][i]=c[best_chromosome][i];                      // copies the best chromosome bit values into the second
chromosome
		temp_c[1]=c[best_chromosome][i];                    // completes the swap by copying the temporary chromosome
	}//end of temp swap to temp_c[]
}


void print_results()
{
	printf("\n After %i iterations, the best chromosomes are :   \n",no_of_runs);
	printf("\n After %i actual iterations, the best chromosomes are :   \n",actual_no_of_runs);
	for (k=0;k<4;k++)
	{
		tv[k]=0.0;                                          // set the total value of each chromosome to zero
		tw[k]=0.0;                                          // set the total weight of each chromosome to zero
		printf("\n [");
		for (i=0;i<10;i++)
		{
			tv[k]=tv[k]+c[k][i]*v[i];                   // cumulative sum of value by bits for each chromosome
			tw[k]=tw[k]+c[k][i]*w[i];                   // cumulative sum of weight by bits for each chromosome
			printf(" %i",(int)c[k][i]);                 // print the final solutions population
		}
		printf(" ]: weight is %f : value is %f\n",tw[k],tv[k]);
	}
	printf("\n");


}
```

Appendix C : Data structure for feature properties with example of 3 rows of data

```
typedef struct st_features
{
	int feat_ID;
	char feat_name[32];
	int feat_RAM_size;
	int feat_ROM_size;
	int feat_EEPROM_size;
	char feat_com_protocol[10];
	int feat_com_speed;
	char feat_com_chip[16];
	int feat_proc_speed;
} st_feature;
```

```
st_feature feature_properties[]={
//	ID	Name		RAM_size	ROM_size	EEPROM_size	protocol	com_speed	com_chip	proc_speed
	0,	"Controller  ",	2648,		4096,		4096,		"CAN",		4096,		"MCP2515",	4096,
	1,	"Calliper    ",	7516,		2048,		4096,		"CAN",		2048,		"MCP2515",	4096,
	2,	"Brake Pedal ",	3886,		1024,		4096,		"CAN",		1024,		"MCP2515",	4096};
```

Appendix D : Data structure for ECU properties with example of 3 rows of data

```
typedef struct st_ECUS
{
	int ECU_ID;
	char ECU_name[32];
	int ECU_RAM_size;
	int ECU_ROM_size;
	int ECU_EEPROM_size;
	char ECU_com_protocol[10];
	int ECU_com_speed;
	char ECU_com_chip[16];
	int ECU_proc_speed;
	int ECU_cost;
} st_ECU;
```

```
st_ECU ECU_properties[] = {
//	ID	Name		RAM_size	ROM_size	EEPROM_size	protocol	com_speed	com_chip	proc_speed	Cost
	0,	"ECU_0",	28564,		400,		4096,		"CAN",		400,		"MCP2515",	400,		1
	1,	"ECU_1",	28564,		400,		4096,		"CAN",		400,		"MCP2515",	400,		1
	2,	"ECU_2",	28564,		400,		4096,		"CAN",		400,		"MCP2515",	400,		1};
```

Appendix E : Full size table of results for GA runs

Table 8-1 : Results of successive executions of GA for 2 to 6 features

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD
1	Feats	ECU	Feat	ROM	Speed	Com	Feats	ECU	Feat	ROM	Speed	Com	Feats	ECU	Feat	ROM	Speed	Com	Feats	ECU	Feat	ROM	Speed	Com	Feats	ECU	Feat	ROM	Speed	Com
2	6	0		80000	30000000	1300	5	0		80000	30000000	1300	4	0		80000	30000000	1300	3	0		80000	30000000	1300	2	0		80000	30000000	1300
3	X	73856	0	1024	1000	250	X	74880	0	1024	1000	250	X	75904	0	1024	1000	250	1	76928	0	1024	1000	250	1	77952	0	1024	1000	250
4		29994000	1	1024	1000	500		29995000	1	1024	1000	500		29996000	1	1024	1000	500		29997000	1	1024	1000	500		29998000	1	1024	1000	500
5		-1450	2	1024	1000	500		-950	2	1024	1000	500		-450	2	1024	1000	500		50	2	1024	1000	500		550	2	1024	1000	500
6			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
7			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
8			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
9	6	1		800000	30000200	12590		1		800000	30000200	12590		1		800000	30000200	12590		1		800000	30000200	12590		1		800000	30000200	12590
10	1	793856	0	1024	1000	250	1	794880	0	1024	1000	250	1	795904	0	1024	1000	250	1	796928	0	1024	1000	250	1	797952	0	1024	1000	250
11		29994200	1	1024	1000	500		29995200	1	1024	1000	500		29996200	1	1024	1000	500		29997200	1	1024	1000	500		29998200	1	1024	1000	500
12		9840	2	1024	1000	500		10340	2	1024	1000	500		10840	2	1024	1000	500		11340	2	1024	1000	500		11840	2	1024	1000	500
13			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
14			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
15			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
16		2		409600	15000000	5000		2		409600	15000000	5000		2		409600	15000000	5000		2		409600	15000000	5000		2		409600	15000000	5000
17	1	403456	0	1024	1000	250	1	404480	0	1024	1000	250	1	405504	0	1024	1000	250	1	406528	0	1024	1000	250	1	407552	0	1024	1000	250
18		14994000	1	1024	1000	500		14995000	1	1024	1000	500		14996000	1	1024	1000	500		14997000	1	1024	1000	500		14998000	1	1024	1000	500
19		2250	2	1024	1000	500		2750	2	1024	1000	500		3250	2	1024	1000	500		3750	2	1024	1000	500		4250	2	1024	1000	500
20			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
21			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
22			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
23		3		409600	12000000	5000		3		409600	12000000	5000		3		409600	12000000	5000		3		409600	12000000	5000		3		409600	12000000	5000
24	1	403456	0	1024	1000	250	1	404480	0	1024	1000	250	1	405504	0	1024	1000	250	1	406528	0	1024	1000	250	1	407552	0	1024	1000	250
25		11994000	1	1024	1000	500		11995000	1	1024	1000	500		11996000	1	1024	1000	500		11997000	1	1024	1000	500		11998000	1	1024	1000	500
26		2250	2	1024	1000	500		2750	2	1024	1000	500		3250	2	1024	1000	500		3750	2	1024	1000	500		4250	2	1024	1000	500
27			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
28			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
29			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
30		4		409600	10000000	5000		4		409600	10000000	5000		4		409600	10000000	5000		4		409600	10000000	5000		4		409600	10000000	5000
31	1	403456	0	1024	1000	250	1	404480	0	1024	1000	250	1	405504	0	1024	1000	250	1	406528	0	1024	1000	250	1	407552	0	1024	1000	250
32		9994000	1	1024	1000	500		9995000	1	1024	1000	500		9996000	1	1024	1000	500		9997000	1	1024	1000	500		9998000	1	1024	1000	500
33		2250	2	1024	1000	500		2750	2	1024	1000	500		3250	2	1024	1000	500		3750	2	1024	1000	500		4250	2	1024	1000	500
34			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
35			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
36			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500
37		5		409600	10000000	5000		5		409600	10000000	5000		5		409600	10000000	5000		5		409600	10000000	5000		5		409600	10000000	5000
38	1	403456	0	1024	1000	250	1	404480	0	1024	1000	250	1	405504	0	1024	1000	250	1	406528	0	1024	1000	250	1	407552	0	1024	1000	250
39		9994000	1	1024	1000	500		9995000	1	1024	1000	500		9996000	1	1024	1000	500		9997000	1	1024	1000	500		9998000	1	1024	1000	500
40		2250	2	1024	1000	500		2750	2	1024	1000	500		3250	2	1024	1000	500		3750	2	1024	1000	500		4250	2	1024	1000	500
41			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500			3	1024	1000	500
42			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500			4	1024	1000	500
43			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500			5	1024	1000	500

Appendix F : Synthetic data for feature properties used to test the GA

```
st_feature feature_properties[]={

//feat_ID   feat_name          feat_RAM_size    *feat_ROM_size   feat_EEPROM_size   feat_com_protocol   *feat_com_speed   feat_com_chip   *feat_proc_speed

0,          "ABS        ",    4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

1,          "Pedal      ",    4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

2,          "IMU        ",    4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

3,          "FL wheel   ",    4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

4,          "FR wheel   ",    4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

5,          "BL wheel   ",    4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

6,          "BR wheel   ",    4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

7,          "FL calliper ",   4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

8,          "FR calliper ",   4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

9,          "BL calliper ",   4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

10,         "BR calliper ",   4096,            100,             4096,              "CAN",              100,              "MCP2515",      100,

11,         "ACC_Pedal  ",    4096,            100,             4096,              "CAN",              100,              "MCP2515",      100};
```

Appendix G : Synthetic data for ECU properties used to test the GA

st_ECU ECU_properties[] = {

//ECU_ID	ECU_name	ECU_RAM_size	*ECU_ROM_size	ECU_EEPROM_size	ECU_com_protocol	*ECU_com_speed	ECU_com_chip	*ECU_proc_speed	ECU_cost
0,	"ECU_0",	4096,	400,	4096,	"CAN",	400,	"MCP2515",	400,	2,
1,	"ECU_1",	4096,	400,	4096,	"CAN",	400,	"MCP2515",	400,	2,
2,	"ECU_2",	4096,	400,	4096,	"CAN",	400,	"MCP2515",	400,	2,
3,	"ECU_3",	4096,	312,	4096,	"CAN",	312,	"MCP2515",	312,	16,
4,	"ECU_4",	4096,	312,	4096,	"CAN",	312,	"MCP2515",	312,	16,
5,	"ECU_5",	4096,	312,	4096,	"CAN",	312,	"MCP2515",	312,	16,
6,	"ECU_6",	4096,	400,	4096,	"CAN",	400,	"MCP2515",	400,	2,
7,	"ECU_7",	4096,	400,	4096,	"CAN",	400,	"MCP2515",	400,	2,
8,	"ECU_8",	4096,	312,	4096,	"CAN",	312,	"MCP2515",	312,	16,
9,	"ECU_9",	4096,	360,	4096,	"CAN",	300,	"MCP2515",	300,	16,
10,	"ECU_10",	4096,	360,	4096,	"CAN",	300,	"MCP2515",	300,	16,
11,	"ECU_11",	4096,	360,	4096,	"CAN",	300,	"MCP2515",	300,	16};

Appendix H : Output of execution of ESP for 12 features

1 run(s) of ECU/FEATURES PROBLEM with 12 features

Chromosome[2] is

```
[
 0  0  0  0  0  0  0  0  1  1  1  1
 0  0  0  0  1  1  1  1  0  0  0  0
 1  1  1  1  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0 ]
```

Global iteration count is 39091131:

ECU no 0 supports feature numbers   : 8 : 9 : 10 : 11
ECU no 1 supports feature numbers   : 4 : 5 : 6 : 7
ECU no 2 supports feature numbers   : 0 : 1 : 2 : 3
Cost of solution is 6

Number of genes selected after validity checks = 39091130:

Global iteration count = 39091130

Highest iteration number is 39091130

Average runtime after total number of runs = 312.0350000

Number of genes selected after validity checks = 39091130:

EXHAUSTIVE MODEL

Reminder: CLOCKS_PER_SEC = 1000

This was run on the desktop machine

Main is ending.

Programme finished!

Appendix I : Code generated by xtUML for methods and test sequences

The following are examples of code generated automatically from the xtUML IDE based on user defined class diagrams and state machines

Machine.c

```
/*-----------------------------------------------------------------------
 * File:  Machine.c
 *
 * UML Component Port Messages
 * Component/Module Name:  Machine
 *
 * your copyright statement can go here (from te_copyright.body)
 *-----------------------------------------------------------------------*/

#include "test_binaries_sys_types.h"
#include "Machine.h"
#include "TIM_bridge.h"
#include "Machine_ARCH_bridge.h"
#include "Machine_classes.h"

/*
 * UML Domain Functions (Synchronous Services)
 */

/*
 * Domain Function:  Decrease_Pedal_Angle
 */
void
Machine_Decrease_Pedal_Angle()
{
  Machine_Pedal * pedal=0;
  /* SELECT any pedal FROM INSTANCES OF Pedal */
  XTUML_OAL_STMT_TRACE( 1, "SELECT any pedal FROM INSTANCES OF Pedal" );
```

```
  pedal = (Machine_Pedal *) Escher_SetGetAny( &pG_Machine_Pedal_extent.active );
  /* GENERATE Pedal2:Decrease_Angle() TO pedal */
  XTUML_OAL_STMT_TRACE( 1, "GENERATE Pedal2:Decrease_Angle() TO pedal" );
  { Escher_xtUMLEvent_t * e = Escher_NewxtUMLEvent( pedal, &Machine_Pedalevent2c );
    Escher_SendEvent( e );
  }

}

/*
 * Domain Function:  DefineABS
 */
void
Machine_DefineABS()
{
  Machine_IMU * imu;Machine_Wheel_Brake * wheel;Machine_Pedal * pedal;Machine_ABS_Control * abs_con;
  /* CREATE OBJECT INSTANCE abs_con OF ABS_Control */
  XTUML_OAL_STMT_TRACE( 1, "CREATE OBJECT INSTANCE abs_con OF ABS_Control" );
  abs_con = (Machine_ABS_Control *) Escher_CreateInstance( Machine_DOMAIN_ID, Machine_ABS_Control_CLASS_NUMBER );
  /* CREATE OBJECT INSTANCE pedal OF Pedal */
  XTUML_OAL_STMT_TRACE( 1, "CREATE OBJECT INSTANCE pedal OF Pedal" );
  pedal = (Machine_Pedal *) Escher_CreateInstance( Machine_DOMAIN_ID, Machine_Pedal_CLASS_NUMBER );
  /* RELATE pedal TO abs_con ACROSS R1 */
  XTUML_OAL_STMT_TRACE( 1, "RELATE pedal TO abs_con ACROSS R1" );
  Machine_ABS_Control_R1_Link( pedal, abs_con );
  /* ASSIGN pedal.Pressure_Angle = minimum */
  XTUML_OAL_STMT_TRACE( 1, "ASSIGN pedal.Pressure_Angle = minimum" );
  pedal->Pressure_Angle = Machine_pedal_angle_minimum_e;
  /* ASSIGN pedal.Feedback_Force = 0 */
  XTUML_OAL_STMT_TRACE( 1, "ASSIGN pedal.Feedback_Force = 0" );
  pedal->Feedback_Force = 0;
  /* CREATE OBJECT INSTANCE wheel OF Wheel_Brake */
  XTUML_OAL_STMT_TRACE( 1, "CREATE OBJECT INSTANCE wheel OF Wheel_Brake" );
  wheel = (Machine_Wheel_Brake *) Escher_CreateInstance( Machine_DOMAIN_ID, Machine_Wheel_Brake_CLASS_NUMBER );
  /* RELATE wheel TO abs_con ACROSS R3 */
  XTUML_OAL_STMT_TRACE( 1, "RELATE wheel TO abs_con ACROSS R3" );
  Machine_ABS_Control_R3_Link( wheel, abs_con );
```

```
  /* ASSIGN wheel.Rotational_Speed = 0 */
  XTUML_OAL_STMT_TRACE( 1, "ASSIGN wheel.Rotational_Speed = 0" );
  wheel->Rotational_Speed = 0;
  /* CREATE OBJECT INSTANCE imu OF IMU */
  XTUML_OAL_STMT_TRACE( 1, "CREATE OBJECT INSTANCE imu OF IMU" );
  imu = (Machine_IMU *) Escher_CreateInstance( Machine_DOMAIN_ID, Machine_IMU_CLASS_NUMBER );
  /* RELATE imu TO abs_con ACROSS R2 */
  XTUML_OAL_STMT_TRACE( 1, "RELATE imu TO abs_con ACROSS R2" );
  Machine_ABS_Control_R2_Link( imu, abs_con );
  /* ASSIGN imu.Long_Val = 0 */
  XTUML_OAL_STMT_TRACE( 1, "ASSIGN imu.Long_Val = 0" );
  imu->Long_Val = 0;
  /* ASSIGN imu.Lat_Val = 0 */
  XTUML_OAL_STMT_TRACE( 1, "ASSIGN imu.Lat_Val = 0" );
  imu->Lat_Val = 0;
  /* ASSIGN imu.Rot_Val = 0 */
  XTUML_OAL_STMT_TRACE( 1, "ASSIGN imu.Rot_Val = 0" );
  imu->Rot_Val = 0;

}

/*
 * Domain Function:  Increase_Pedal_Angle
 */
void
Machine_Increase_Pedal_Angle()
{
  Machine_Pedal * pedal=0;
  /* SELECT any pedal FROM INSTANCES OF Pedal */
  XTUML_OAL_STMT_TRACE( 1, "SELECT any pedal FROM INSTANCES OF Pedal" );
  pedal = (Machine_Pedal *) Escher_SetGetAny( &pG_Machine_Pedal_extent.active );
  /* GENERATE Pedal1:Increase_Angle() TO pedal */
  XTUML_OAL_STMT_TRACE( 1, "GENERATE Pedal1:Increase_Angle() TO pedal" );
  { Escher_xtUMLEvent_t * e = Escher_NewxtUMLEvent( pedal, &Machine_Pedalevent1c );
    Escher_SendEvent( e );
  }
```

```
}

/*
 * Domain Function:  TestSequence1
 */
void
Machine_TestSequence1()
{
  Machine_TestSequences * testSequence;
  /* CREATE OBJECT INSTANCE testSequence OF TestSequences */
  XTUML_OAL_STMT_TRACE( 1, "CREATE OBJECT INSTANCE testSequence OF TestSequences" );
  testSequence = (Machine_TestSequences *) Escher_CreateInstance( Machine_DOMAIN_ID, Machine_TestSequences_CLASS_NUMBER );
  /* GENERATE TestSequences2:perform_test_seq_1() TO testSequence */
  XTUML_OAL_STMT_TRACE( 1, "GENERATE TestSequences2:perform_test_seq_1() TO testSequence" );
  { Escher_xtUMLEvent_t * e = Escher_NewxtUMLEvent( testSequence, &Machine_TestSequencesevent2c );
    Escher_SendEvent( e );
  }

}

/* xtUML class info (collections, sizes, etc.) */
Escher_Extent_t * const Machine_class_info[ Machine_MAX_CLASS_NUMBERS ] = {
  &pG_Machine_TestSequences_extent,
  &pG_Machine_Pedal_extent,
  &pG_Machine_Con_Pedal_extent,
  &pG_Machine_ABS_Control_extent,
  &pG_Machine_Wheel_Brake_extent,
  &pG_Machine_IMU_extent
};

/*
 * Array of pointers to the class event dispatcher method.
 * Index is the (model compiler enumerated) number of the state model.
 */
const EventTaker_t Machine_EventDispatcher[ Machine_STATE_MODELS ] = {
  Machine_class_dispatchers
};
```

```
void Machine_execute_initialization()
{
  /*
   * Initialization Function:  DefineABS
   * Component:  Machine
   */
  Machine_DefineABS();

  /*
   * Initialization Function:  TestSequence1
   * Component:  Machine
   */
  Machine_TestSequence1();

}
```

Header file “Machine_ABS_Control_Class.h”

```
/*-----------------------------------------------------------------------------
 * File:  Machine_ABS_Control_class.h
 *
 * Class:       ABS_Control  (ABS_Control)
 * Component:   Machine
 *
 * your copyright statement can go here (from te_copyright.body)
 *----------------------------------------------------------------------------*/

#ifndef MACHINE_ABS_CONTROL_CLASS_H
#define MACHINE_ABS_CONTROL_CLASS_H

#ifdef __cplusplus
extern "C"    {
#endif

/*
```

```
 * Structural representation of application analysis class:
 *   ABS_Control  (ABS_Control)
 */
struct Machine_ABS_Control {

  /* application analysis class attributes */
  Escher_UniqueID_t Control_ID;  /* - Control_ID */
  Escher_UniqueID_t IMU_ID;  /* - IMU_ID */
  Escher_UniqueID_t Pedal_ID;  /* - Pedal_ID */
  Escher_UniqueID_t Wheel_Brake_ID;  /* - Wheel_Brake_ID */

  /* relationship storage */
  /* Note:  No storage needed for ABS_Control->Pedal[R1] */
  /* Note:  No storage needed for ABS_Control->IMU[R2] */
  /* Note:  No storage needed for ABS_Control->Wheel_Brake[R3] */
};
void Machine_ABS_Control_op_Apply_Brake_Pressure( Machine_ABS_Control * );
void Machine_ABS_Control_op_Monitor_Wheel_Speed( Machine_ABS_Control * );
void Machine_ABS_Control_op_Release_Brake_Pressure( Machine_ABS_Control * );
void Machine_ABS_Control_op_Monitor_Road_Speed( Machine_ABS_Control * );
void Machine_ABS_Control_op_Receive_Pedal_Signal( Machine_ABS_Control * );
void Machine_ABS_Control_op_Monitor_Lateral_Acceleration( Machine_ABS_Control * );
void Machine_ABS_Control_op_Monitor_Yaw_Angle( Machine_ABS_Control * );
void Machine_ABS_Control_R1_Link( Machine_Pedal *, Machine_ABS_Control * );
/* Note:  Pedal<-R1->ABS_Control unrelate accessor not needed */
void Machine_ABS_Control_R2_Link( Machine_IMU *, Machine_ABS_Control * );
/* Note:  IMU<-R2->ABS_Control unrelate accessor not needed */
void Machine_ABS_Control_R3_Link( Machine_Wheel_Brake *, Machine_ABS_Control * );
/* Note:  Wheel_Brake<-R3->ABS_Control unrelate accessor not needed */


#define Machine_ABS_Control_MAX_EXTENT_SIZE 10
extern Escher_Extent_t pG_Machine_ABS_Control_extent;

#ifdef __cplusplus
}
#endif
```

```
#endif  /* MACHINE_ABS_CONTROL_CLASS_H */
//Create the instances in the system.


//Create the ABS Controller ECU
create object instance abs_con of ABS_Control;

//assign Pedal.Pressure_Angle = pedal_angle::minimum;

//Create the Brake Pedal ECU
create object instance pedal of Pedal;
relate pedal to abs_con across R1;
assign pedal.Pressure_Angle = pedal_angle::minimum;
assign pedal.Feedback_Force = 0;

//Create the Wheel/Brake ECU
create object instance wheel of Wheel_Brake;
relate wheel to abs_con across R3;
assign wheel.Rotational_Speed = 0;

//Create the IMU ECU
create object instance imu of IMU;
relate imu to abs_con across R2;
assign imu.Long_Val = 0;
assign imu.Lat_Val = 0;
assign imu.Rot_Val = 0;
```

Appendix J : Real time output of ABS/AMT Callipers Throttle Wheels ECU

=~=~=~=~=~=~=~=~=~=~=~=~=~= PuTTY log 2019.06.25 21:48:19 =~=~=~=~=~=~=~=~=~=~=~=~=~=

Starting set up for Callipers Throttle Wheels program (COM 6)

CAN Init OK.

THROTTLE POTENTIOMETER present for use : value was 0

Wheel Speeds have received a message from Phys Mod on the CAN bus
0.00000 0.00000 0.00000 0.00000
3.35393 3.35393 3.35393 3.35393
6.70787 6.70787 6.70787 6.70787
10.54094 10.54094 10.54094 10.54094
15.81293 15.81293 15.81293 15.81293
21.08340 21.08340 21.08340 21.08340
22.04166 22.04166 22.04166 22.04166
27.55323 27.55323 27.55323 27.55323
33.06326 33.06326 33.06326 33.06326
46.00139 46.00139 46.00139 46.00139
46.00139 46.00139 46.00139 46.00139
46.00139 46.00139 46.00139 46.00139
53.66905 53.66905 53.66905 53.66905
90.56690 90.56690 90.56690 90.56690
77.62877 77.62877 77.62877 77.62877
90.56690 90.56690 90.56690 90.56690

// brake demand message has been received via CAN
Calliper values have changed : send via CAN and call the phys mod function
Wheel Speeds have received a message from Phys Mod on the CAN bus
82.56660 82.56660 82.56660 82.56660

Calliper values have changed : send via CAN and call the phys mod function
Wheel Speeds have received a message from Phys Mod on the CAN bus
74.56629 74.56629 74.56629 74.56629

Calliper values have changed : send via CAN and call the phys mod function
Wheel Speeds have received a message from Phys Mod on the CAN bus
66.56598 66.56598 66.56598 66.56598

Calliper values have changed : send via CAN and call the phys mod function
Wheel Speeds have received a message from Phys Mod on the CAN bus
34.56627 34.56627 34.56627 34.56627

Calliper values have changed : send via CAN and call the phys mod function
Wheel Speeds have received a message from Phys Mod on the CAN bus
10.56688 10.56688 10.56688 10.56688

Calliper values have changed : send via CAN and call the phys mod function
Wheel Speeds have received a message from Phys Mod on the CAN bus
2.56657 2.56657 2.56657 2.56657

Calliper values have changed : send via CAN and call the phys mod function
Wheel Speeds have received a message from Phys Mod on the CAN bus
0.00000 0.00000 0.00000 0.00000

Appendix K : Real time output of ABS/AMT Physics Model & Environment

=~=~=~=~=~=~=~=~=~=~=~=~=~= PuTTY log 2019.06.25 21:47:29 =~=~=~=~=~=~=~=~=~=~=~=~=~=

Starting set up PHYSICS and ENVIRONMENT VARIABLES program (COM 8)

CAN Init OK.

SENSOR VALUE IS 1023 at start up

ENVIRONMENT POTENTIOMETER present for use : value was 1023

Environment values are: : 0 : 0 : 0 : 0 // hard coded initialisation in case potentiometer not present

Environment_values are: : 8 : 8 : 8 : 8 // Default starting values for friction coefficient
Environment_values are: : 7 : 7 : 7 : 7 // User drives friction values from potentiometer to test input
Environment_values are: : 6 : 6 : 6 : 6
Environment_values are: : 5 : 5 : 5 : 5
Environment_values are: : 4 : 4 : 4 : 4
Environment_values are: : 3 : 3 : 3 : 3
Environment_values are: : 2 : 2 : 2 : 2
Environment_values are: : 1 : 1 : 1 : 1
Environment_values are: : 2 : 2 : 2 : 2
Environment_values are: : 3 : 3 : 3 : 3
Environment_values are: : 4 : 4 : 4 : 4
Environment_values are: : 5 : 5 : 5 : 5
Environment_values are: : 6 : 6 : 6 : 6
Environment_values are: : 7 : 7 : 7 : 7
Environment_values are: : 8 : 8 : 8 : 8

MESSAGE RECEIVED FROM ID 60 // gear stick
MESSAGE RECEIVED FROM ID 80 // AMT
MESSAGE RECEIVED FROM ID 180 // gear number

gear number received at Phys mod // user has selected gear upshift from gear stick ECU via CAN
Gear number is 1
Road speed from gear number is 0.00000 // no road speed yet as throttle has not been applied
Sent Wheel Speeds message via CAN. Sent values : : 0 : 0

ENGINE MESSAGE TO PHYS MOD //phys mod receives throttle message via CAN
Revs : gear : g_road_speed
929 1 3.35435

Buffered for CAN road speed from throttle number is 2198

ENGINE MESSAGE TO PHYS MOD //throttle is increased
Revs : gear : g_road_speed
1858 1 6.70870

Buffered for CAN road speed from throttle number is 4396

gear number received at Phys mod //another upshift is requested on the gear stick module
Gear number is 2
Road speed from gear number is 10.54224
Sent Wheel Speeds message via CAN. Sent values : : 252 : 26

ENGINE MESSAGE TO PHYS MOD // throttle is increased again
Revs : gear : g_road_speed
2787 2 15.81335

Buffered for CAN road speed from throttle number is 10363

ENGINE MESSAGE TO PHYS MOD //another throttle increase
Revs : gear : g_road_speed
3716 2 21.08447

Buffered for CAN road speed from throttle number is 13817

gear number received at Phys mod //another gear shift requested from gear stick
Gear number is 3
Road speed from gear number is 22.04286
Sent Wheel Speeds message via CAN. Sent values : : 109 : 56

ENGINE MESSAGE TO PHYS MOD //another increase in throttle to engine revs
Revs : gear : g_road_speed
4645 3 27.55357

Buffered for CAN road speed from throttle number is 18057

ENGINE MESSAGE TO PHYS MOD //another increase in throttle
Revs : gear : g_road_speed
5574 3 33.06428

Buffered for CAN road speed from throttle number is 21668

gear number received at Phys mod //upshift selected from gear stick
Gear number is 4
Road speed from gear number is 46.00248
Sent Wheel Speeds message via CAN. Sent values : : 195 : 117

ENGINE MESSAGE TO PHYS MOD // another upshift
Revs : gear : g_road_speed
4645 4 38.33540

Buffered for CAN road speed from throttle number is 25123

ENGINE MESSAGE TO PHYS MOD //another throttle increase
Revs : gear : g_road_speed
5574 4 46.00248
Buffered for CAN road speed from throttle number is 30147

ENGINE MESSAGE TO PHYS MOD
Revs : gear : g_road_speed
5574 4 46.00248

Buffered for CAN road speed from throttle number is 30147

ENGINE MESSAGE TO PHYS MOD // throttle increase to maximum revs
Revs : gear : g_road_speed
6503 4 53.66956

Buffered for CAN road speed from throttle number is 35172

gear number received at Phys mod //upshift to top gear and top speed
Gear number is 5
Road speed from gear number is 90.56739
Sent Wheel Speeds message via CAN. Sent values : : 217 : 231

ENGINE MESSAGE TO PHYS MOD // accidental blip of throttle reduces speed
Revs : gear : g_road_speed
5574 5 77.62919

Buffered for CAN road speed from throttle number is 50874

ENGINE MESSAGE TO PHYS MOD // back to full revs
Revs : gear : g_road_speed
6503 5 90.56739

Buffered for CAN road speed from throttle number is 59353

Reached phys mod with message from callipers. Why nothing happening?

Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

We're in the physics_model CPP program

Road Speed before being assigned to pm_IMU_longitudinal 90.56739

Setting wheel speed in X2 in = 90.57
Setting wheel speed in X2 out = 82.57
Setting wheel speed in X2 in = 90.57
Setting wheel speed in X2 out = 82.57
Setting wheel speed in X2 in = 90.57
Setting wheel speed in X2 out = 82.57
Setting wheel speed in X2 in = 90.57
Setting wheel speed in X2 out = 82.57

ARRAY of pm_wheel_speeds 82.56690 82.56690 82.56690 82.56690

GLOBAL wheel speeds 82.56690 82.56690 82.56690 82.56690

AVERAGE STOPPING FORCE * TIME SLICE 8.00000

PHYSICS MODEL FUNCTION IS ATTEMPTING TO RETURN ROAD SPEED VALUE : 82.56739

wheels speeds that will be returned:82.56690 82.56690 82.56690 82.56690

Deliberately pausing to give the real time effect of the time taken by the physics model

Actually paused for 1000028 microseconds

pm_physics_model_func has returned IMU = 82.57
pm_physics_model_func has returned wheel speeds = 82.57

Brake pedal received at Phys mod
Reached phys mod with message from callipers.

Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

We're in the physics_model CPP program

Road Speed before being assigned to pm_IMU_longitudinal 82.56739

Setting wheel speed in X2 in = 82.57
Setting wheel speed in X2 out = 74.57
Setting wheel speed in X2 in = 82.57
Setting wheel speed in X2 out = 74.57
Setting wheel speed in X2 in = 82.57
Setting wheel speed in X2 out = 74.57
Setting wheel speed in X2 in = 82.57
Setting wheel speed in X2 out = 74.57

ARRAY of pm_wheel_speeds 74.56690 74.56690 74.56690 74.56690

GLOBAL wheel speeds 74.56690 74.56690 74.56690 74.56690

AVERAGE STOPPING FORCE * TIME SLICE 8.00000

PHYSICS MODEL FUNCTION IS ATTEMPTING TO RETURN ROAD SPEED VALUE : 74.56739

wheels speeds that will be returned:74.56690 74.56690 74.56690 74.56690

Deliberately pausing to give the real time effect of the time taken by the physics model

Actually paused for 1000020 microseconds

pm_physics_model_func has returned IMU = 74.57
pm_physics_model_func has returned wheel speeds = 74.57

Brake pedal received at Phys mod : pedal being pressed is 0

Reached phys mod with message from callipers

Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

We're in the physics_model CPP program

Road Speed before being assigned to pm_IMU_longitudinal 74.56739

Setting wheel speed in X2 in = 74.57
Setting wheel speed in X2 out = 66.57
Setting wheel speed in X2 in = 74.57
Setting wheel speed in X2 out = 66.57
Setting wheel speed in X2 in = 74.57
Setting wheel speed in X2 out = 66.57
Setting wheel speed in X2 in = 74.57
Setting wheel speed in X2 out = 66.57

ARRAY of pm_wheel_speeds 66.56690 66.56690 66.56690 66.56690

GLOBAL wheel speeds 66.56690 66.56690 66.56690 66.56690

AVERAGE STOPPING FORCE * TIME SLICE 8.00000

PHYSICS MODEL FUNCTION IS ATTEMPTING TO RETURN ROAD SPEED VALUE : 66.56739

wheels speeds that will be returned:66.56690 66.56690 66.56690 66.56690

Deliberately pausing to give the real time effect of the time taken by the physics model

Actually paused for 1000028 microseconds

pm_physics_model_func has returned IMU = 66.57
pm_physics_model_func has returned wheel speeds = 66.57

Brake pedal received at Phys mod : pedal being pressed is 0

Reached phys mod with message from callipers

Brake Calliper values in int and double are
4 4.00
4 4.00
4 4.00

4 4.00

We're in the physics_model CPP program

Road Speed before being assigned to pm_IMU_longitudinal 66.56739

Setting wheel speed in X2 in = 66.57
Setting wheel speed in X2 out = 34.57
Setting wheel speed in X2 in = 66.57
Setting wheel speed in X2 out = 34.57
Setting wheel speed in X2 in = 66.57
Setting wheel speed in X2 out = 34.57
Setting wheel speed in X2 in = 66.57
Setting wheel speed in X2 out = 34.57

ARRAY of pm_wheel_speeds 34.56690 34.56690 34.56690 34.56690

GLOBAL wheel speeds 34.56690 34.56690 34.56690 34.56690

AVERAGE STOPPING FORCE * TIME SLICE 32.00000

PHYSICS MODEL FUNCTION IS ATTEMPTING TO RETURN ROAD SPEED VALUE : 34.56739

wheels speeds that will be returned:34.56690 34.56690 34.56690 34.56690

Deliberately pausing to give the real time effect of the time taken by the physics model

Actually paused for 1000024 microseconds

pm_physics_model_func has returned IMU = 34.57
pm_physics_model_func has returned wheel speeds = 34.57

Brake pedal received at Phys mod

Reached phys mod with message from callipers

Brake Calliper values in int and double are
3 3.00
3 3.00
3 3.00
3 3.00

We're in the physics_model CPP program

Road Speed before being assigned to pm_IMU_longitudinal 34.56739

Setting wheel speed in X2 in = 34.57
Setting wheel speed in X2 out = 10.57
Setting wheel speed in X2 in = 34.57
Setting wheel speed in X2 out = 10.57
Setting wheel speed in X2 in = 34.57

Setting wheel speed in X2 out = 10.57
Setting wheel speed in X2 in = 34.57
Setting wheel speed in X2 out = 10.57

ARRAY of pm_wheel_speeds 10.56690 10.56690 10.56690 10.56690

GLOBAL wheel speeds 10.56690 10.56690 10.56690 10.56690

AVERAGE STOPPING FORCE * TIME SLICE 24.00000

PHYSICS MODEL FUNCTION IS ATTEMPTING TO RETURN ROAD SPEED VALUE : 10.56739

wheels speeds that will be returned:10.56690 10.56690 10.56690 10.56690

Deliberately pausing to give the real time effect of the time taken by the physics model

Actually paused for 1000020 microseconds

pm_physics_model_func has returned IMU = 10.57
pm_physics_model_func has returned wheel speeds = 10.57

Brake pedal received at Phys mod : pedal being pressed is 0

Reached phys mod with message from callipers

Brake Calliper values in int and double are
1   1.00
1   1.00
1   1.00
1   1.00

We're in the physics_model CPP program

Road Speed before being assigned to pm_IMU_longitudinal 10.56739

Setting wheel speed in X2 in = 10.57
Setting wheel speed in X2 out = 2.57
Setting wheel speed in X2 in = 10.57
Setting wheel speed in X2 out = 2.57
Setting wheel speed in X2 in = 10.57
Setting wheel speed in X2 out = 2.57
Setting wheel speed in X2 in = 10.57
Setting wheel speed in X2 out = 2.57

ARRAY of pm_wheel_speeds 2.56690 2.56690 2.56690 2.56690

GLOBAL wheel speeds 2.56690 2.56690 2.56690 2.56690

AVERAGE STOPPING FORCE * TIME SLICE 8.00000

PHYSICS MODEL FUNCTION IS ATTEMPTING TO RETURN ROAD SPEED VALUE : 2.56739

wheels speeds that will be returned:2.56690 2.56690 2.56690 2.56690

Deliberately pausing to give the real time effect of the time taken by the physics model

Actually paused for 1000024 microseconds

pm_physics_model_func has returned IMU = 2.57
pm_physics_model_func has returned wheel speeds = 2.57

Brake pedal received at Phys mod : pedal being pressed is 0

Reached phys mod with message from callipers

Brake Calliper values in int and double are
1   1.00
1   1.00
1   1.00
1   1.00

We're in the physics_model CPP program

Road Speed before being assigned to pm_IMU_longitudinal 2.56739

About to set wheel speed from 2.57
Setting wheel speed in X1 1.00
About to set wheel speed from 2.57
Setting wheel speed in X1 1.00
About to set wheel speed from 2.57
Setting wheel speed in X1 1.00
About to set wheel speed from 2.57
Setting wheel speed in X1 1.00

ARRAY of pm_wheel_speeds 0.00000 0.00000 0.00000 0.00000

GLOBAL wheel speeds 0.00000 0.00000 0.00000 0.00000

WHEEL SPEEDS ARE ZERO AND IMU IS LESS THAN ONE TIME SLICE: IMU is
2.56739

ROAD SPEED VALUE IS ZERO

Setting wheel speed in X5

wheels speeds that will be returned:0.00000 0.00000 0.00000 0.00000

Appendix L : Real time output from Brake pedal AMT and Clutch program

=~=~=~=~=~=~=~=~=~=~=~=~=~= PuTTY log 2019.06.25 21:49:17 =~=~=~=~=~=~=~=~=~=~=~=~=~=

Starting set up BRAKE PEDAL program (COM 10)

CAN Init OK.

BRAKE POTENTIOMETER present for use : value was 0

Brake value is:0

Message to AMT from gear stick: msg ID = 3c //gear shifts from gear stick come via here

A change gear request has been received UP AMT sending UPSHIFT data to gear box via CAN

Message to AMT from gear stick: msg ID = 3c

A change gear request has been received UP AMT sending UPSHIFT data to gear box via CAN

Message to AMT from gear stick: msg ID = 3c

A change gear request has been received UP AMT sending UPSHIFT data to gear box via CAN

Message to AMT from gear stick: msg ID = 3c

A change gear request has been received UP AMT sending UPSHIFT data to gear box via CAN

Message to AMT from gear stick: msg ID = 3c

A change gear request has been received UP AMT sending UPSHIFT data to gear box via CAN

//five consecutive upshifts are received from the gear shift and reflected in Physics Model

The value that will be sent as a message via CAN as BP is : 1
The value that will be sent as a message via CAN as char is : 1 0
The value that will be sent as a message via CAN as int is : 1

The value that will be sent as a message via CAN as BP is : 0
The value that will be sent as a message via CAN as char is : 0 0
The value that will be sent as a message via CAN as int is : 0

The value that will be sent as a message via CAN as BP is : 1
The value that will be sent as a message via CAN as char is : 1 0
The value that will be sent as a message via CAN as int is : 1

The value that will be sent as a message via CAN as BP is : 2
The value that will be sent as a message via CAN as char is : 2 0
The value that will be sent as a message via CAN as int is : 2

The value that will be sent as a message via CAN as BP is : 1
The value that will be sent as a message via CAN as char is : 1 0
The value that will be sent as a message via CAN as int is : 1

The value that will be sent as a message via CAN as BP is : 0
The value that will be sent as a message via CAN as char is : 0 0
The value that will be sent as a message via CAN as int is : 0

The value that will be sent as a message via CAN as BP is : 1
The value that will be sent as a message via CAN as char is : 1 0
The value that will be sent as a message via CAN as int is : 1

The value that will be sent as a message via CAN as BP is : 2
The value that will be sent as a message via CAN as char is : 2 0
The value that will be sent as a message via CAN as int is : 2

The value that will be sent as a message via CAN as BP is : 3
The value that will be sent as a message via CAN as char is : 3 0
The value that will be sent as a message via CAN as int is : 3

The value that will be sent as a message via CAN as BP is : 4
The value that will be sent as a message via CAN as char is : 4 0
The value that will be sent as a message via CAN as int is : 4

The value that will be sent as a message via CAN as BP is : 3
The value that will be sent as a message via CAN as char is : 3 0
The value that will be sent as a message via CAN as int is : 3

The value that will be sent as a message via CAN as BP is : 1
The value that will be sent as a message via CAN as char is : 1 0
The value that will be sent as a message via CAN as int is : 1

The value that will be sent as a message via CAN as BP is : 0
The value that will be sent as a message via CAN as char is : 0 0
The value that will be sent as a message via CAN as int is : 0

The value that will be sent as a message via CAN as BP is : 1
The value that will be sent as a message via CAN as char is : 1 0
The value that will be sent as a message via CAN as int is : 1

The value that will be sent as a message via CAN as BP is : 2
The value that will be sent as a message via CAN as char is : 2 0
The value that will be sent as a message via CAN as int is : 2

The value that will be sent as a message via CAN as BP is : 0
The value that will be sent as a message via CAN as char is : 0 0
The value that will be sent as a message via CAN as int is : 0

The value that will be sent as a message via CAN as BP is : 1
The value that will be sent as a message via CAN as char is : 1 0

The value that will be sent as a message via CAN as int is : 1

The value that will be sent as a message via CAN as BP is : 2
The value that will be sent as a message via CAN as char is : 2 0
The value that will be sent as a message via CAN as int is : 2

The value that will be sent as a message via CAN as BP is : 1
The value that will be sent as a message via CAN as char is : 1 0
The value that will be sent as a message via CAN as int is : 1

The value that will be sent as a message via CAN as BP is : 0
The value that will be sent as a message via CAN as char is : 0 0
The value that will be sent as a message via CAN as int is : 0

Appendix M : Real time output of ABS/AMT Engine Gearbox Gearstick IMU

=~=~=~=~=~=~=~=~=~=~=~=~=~= PuTTY log 2019.06.25 21:50:08 =~=~=~=~=~=~=~=~=~=~=~=~=~=

Starting set up program for engine gearbox gearstick IMU (COM 17)

CAN Init OK.

GEARSTICK POTENTIOMETER present for use : value was 0

Gear value is:0

Gear change has been called by the gear stick and sent from this module 1
Message received by GEARBOX from AMT //upshift acknowledged but engine revs still zero
UP to gear number 1

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 0
IMU speed = 0.00000

Message value into ENGINE from THROTTLE is : 1 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 929

Message value into ENGINE from THROTTLE is : 2 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 1858

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 2198
IMU speed = 3.35393

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 4396
IMU speed = 6.70787

Gear change has been called by the gear stick and sent from this module 2
Message received by GEARBOX from AMT
UP to gear number 2

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 6908
IMU speed = 10.54094

Message value into ENGINE from THROTTLE is : 3 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 2787

Message value into ENGINE from THROTTLE is : 4 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 3716

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 10363

IMU speed = 15.81293

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 13817
IMU speed = 21.08340

Gear change has been called by the gear stick and sent from this module 3
Message received by GEARBOX from AMT
UP to gear number 3

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 14445
IMU speed = 22.04166

Message value into ENGINE from THROTTLE is : 5 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 4645

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 18057
IMU speed = 27.55323

Message value into ENGINE from THROTTLE is : 6 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 5574

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 21668
IMU speed = 33.06326

Gear change has been called by the gear stick and sent from this module 4
Message received by GEARBOX from AMT
UP to gear number 4

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 30147
IMU speed = 46.00139

Message value into ENGINE from THROTTLE is : 5 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 4645

Message value into ENGINE from THROTTLE is : 6 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 5574

Message value into ENGINE from THROTTLE is : 5 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 4645

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 25123
IMU speed = 38.33525

Message value into ENGINE from THROTTLE is : 6 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 5574

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 30147
IMU speed = 46.00139

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 30147
IMU speed = 46.00139

Message value into ENGINE from THROTTLE is : 7 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 6503

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 35172
IMU speed = 53.66905

Gear change has been called by the gear stick and sent from this module 5
Message received by GEARBOX from AMT
UP to gear number 5

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 59353
IMU speed = 90.56690

// two upshifts requested while already in top gear
gear change direction is beyond current gear range - selection ignored
gear change direction is beyond current gear range - selection ignored

Message value into ENGINE from THROTTLE is : 6 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 5574

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 50874
IMU speed = 77.62877

Message value into ENGINE from THROTTLE is : 7 0
RCVD engine revs that should be sent as a message via CAN for ABS and AMT is : 6503

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 59353
IMU speed = 90.56690

CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

Brake pedal received at ABS : pedal being pressed is 1

CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
0 0.00
0 0.00
0 0.00
0 0.00

Brake pedal received at ABS : pedal being pressed is 0

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 54110
IMU speed = 82.56660

CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
2 2.00
2 2.00
2 2.00
2 2.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
0 0.00
0 0.00
0 0.00
0 0.00

Brake pedal received at ABS : pedal being pressed is 0

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 48867
IMU speed = 74.56629

CAN message to callipers sent from here directly as brake pedal values

Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
2 2.00
2 2.00
2 2.00
2 2.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
3 3.00
3 3.00
3 3.00
3 3.00

Brake pedal received at ABS : pedal being pressed is 1

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 43624
IMU speed = 66.56598

CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
4 4.00
4 4.00
4 4.00
4 4.00

Brake pedal received at ABS : pedal being pressed is 1

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 22653
IMU speed = 34.56627

CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
3 3.00
3 3.00
3 3.00
3 3.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are

1 1.00
1 1.00
1 1.00
1 1.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
0 0.00
0 0.00
0 0.00
0 0.00

Brake pedal received at ABS : pedal being pressed is 0

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 6925
IMU speed = 10.56688

CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
2 2.00
2 2.00
2 2.00
2 2.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
0 0.00
0 0.00
0 0.00
0 0.00

Brake pedal received at ABS : pedal being pressed is 0

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 1682
IMU speed = 2.56657

CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
1 1.00

1 1.00
1 1.00
1 1.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
2 2.00
2 2.00
2 2.00
2 2.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
1 1.00
1 1.00
1 1.00
1 1.00

Brake pedal received at ABS : pedal being pressed is 1
CAN message to callipers sent from here directly as brake pedal values
Brake Calliper values in int and double are
0 0.00
0 0.00
0 0.00
0 0.00

Brake pedal received at ABS : pedal being pressed is 0

Message received by IMU from Phys mod
g_road_speed from Phys Mod = 0
IMU speed = 0.00000

Appendix N : Source code automatically generated from xtUML class diagrams

The following code was automatically generated from xtUML class diagrams of the ABS system incorporating pedal, IMU, wheel_brake and ABS_Control classes. This is from the file 'Machine_ABS_Control.c' and the stubs clearly show the names of the methods included in the ABS_Control class with empty code statements enclosed in curly brackets

```
/*
 * instance operation:  Apply_Brake_Pressure
 */
void
Machine_ABS_Control_op_Apply_Brake_Pressure( Machine_ABS_Control * self)
{

}

/*
 * instance operation:  Monitor_Wheel_Speed
 */
void
Machine_ABS_Control_op_Monitor_Wheel_Speed( Machine_ABS_Control * self)
{

}

/*
 * instance operation:  Release_Brake_Pressure
 */
void
Machine_ABS_Control_op_Release_Brake_Pressure( Machine_ABS_Control * self)
{

}

/*
 * instance operation:  Monitor_Road_Speed
 */
void
Machine_ABS_Control_op_Monitor_Road_Speed( Machine_ABS_Control * self)
{

}

/*
 * instance operation:  Receive_Pedal_Signal
 */
void
Machine_ABS_Control_op_Receive_Pedal_Signal( Machine_ABS_Control * self)
{

}

/*
 * instance operation:  Monitor_Lateral_Acceleration
 */
void
```

```
Machine_ABS_Control_op_Monitor_Lateral_Acceleration( Machine_ABS_Control * self)
{

}

/*
 * instance operation:  Monitor_Yaw_Angle
 */
void
Machine_ABS_Control_op_Monitor_Yaw_Angle( Machine_ABS_Control * self)
{

}
```

In the file

```
/* Provide definitions of the shapes of the class structures.  */

typedef struct Machine_TestSequences Machine_TestSequences;
typedef struct Machine_Pedal Machine_Pedal;
typedef struct Machine_Con_Pedal Machine_Con_Pedal;
typedef struct Machine_ABS_Control Machine_ABS_Control;
typedef struct Machine_Wheel_Brake Machine_Wheel_Brake;
typedef struct Machine_IMU Machine_IMU;
```

Other files created by the source code automation process included

```
classes.h
Machine_ABS_control.c
Machine_ABS_control.h
Machine_calliper_class.c
Machine_calliper_class.h
Machine_pedal_class.c
Machine_pedal_class.h
Machine_IMU_class.c
Machine_IMU_class.h
Machine_wheel_brake_class.c
Machine_wheel_brake_class.h
```

Appendix O : List of abbreviated terms

Abbreviation	Meaning
ABS	Anti-lock Braking System
ADAS	Advanced Driver-Assistance Systems
ADC	analogue to digital converter
ALU	arithmetic logic unit
AMT	Automated Manual Transmission
ANSI	American National Standards Institute
ATL	Atlas Transformation Language
AUTOSAR	Automotive Open System Architecture
CAN	Controller Area Network
CDP	Component Deployment Problem
CPU	Central Processing Unit
DAC	digital to analogue converter
EA	Evolutionary Algorithms
ECU	Electronic Control Unit
ESP	Exhaustive Search Program
ETLA	Extended TLA (four letter abbreviation)
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GP	Genetic Programming
HABTM	HasAndBelongsToMany
HCI	Human Computer Interface
HILs	Hardware in the Loop system
HMI	Human Machine Interface
HPC	High Performance Computing
IDE	integrated development environment
IMU	Inertial Measurement Unit
ISA	improved simulated annealing
kB	kilobyte
kbit	kilobit
LIN	Local Interconnect Network
MARTE	Modeling and Analysis of Real Time Embedded Systems
Mbit	megabit
µC	Microcontroller
MOF	Meta Object Facility
MOST	Media Oriented Systems Transport
NLP	natural language processing
OCL	Object Constraint Language
OEM	Original Equipment Manufacturer
OMG	Object Management Group
RAM	Random Access Memory
ROM	Read Only Memory

RTaW	RealTime-at-Work
SoC	systems-on-chips
SoS	System of Systems
SUVAT	displacement (s), initial velocity (u), final velocity (v), acceleration (a), time (t)
TLA	Three letter abbreviation
TSP	Travelling Salesman Problem
TTCAN	Time Triggered CAN
UML	Unified Modelling Language
VFB	virtual function bus
VHDL	Very High speed integrated circuit hardware Description Language
ViCAN	Vehicular Wireless CAN
XMI	XML Metadata Interchange
XML	Extensible Markup Language
xtUML	Executable Translatable UML

Appendix P : 'C++' source code for generating partitions of integers

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include<iostream>
using namespace std;

int total_count=0; // gives the total number of runs through the loop and therefore the number of partitions

// A utility function to print an array p[] of size 'n'
void printArray(int p[], int n)
{
   for (int i = 0; i < n; i++)
         {
                  cout << p[i] << " ";
         }
   cout << endl;
}

void printAllUniqueParts(int n)
{
   int p[102]; // An array with more than enough memory to store a partition of up to 100
   int k = 0;  // Index of last element in a partition
   p[k] = n;  // Initialize first partition as number itself
   // This loop first prints current partition, then generates next
   // partition. The loop stops when the current partition has all 1s
   while (true)
   {
      // print current partition
      printArray(p, k+1);
                  total_count++;

      // Generate next partition

      // Find the rightmost non-one value in p[]. Also, update the
      // rem_val so that we know how much value can be accommodated
                  //printf("\nSet rem_val to zero\n");
      int rem_val = 0;
      while (k >= 0 && p[k] == 1)
      {
         rem_val += p[k];
         k--;
      }

      // if k < 0, all the values are 1 so there are no more partitions
      if (k < 0)  return;
```

```
        // Decrease the p[k] found above and adjust the rem_val
                    p[k]--;
        rem_val++;

        // If rem_val is more, then the sorted order is violated.  Divide
        // rem_val in different values of size p[k] and copy these values at
        // different positions after p[k]
        while (rem_val > p[k])
        {
                                p[k+1] = p[k];
           rem_val = rem_val - p[k];
           k++;
        }

        //Copy rem_val to next position and increment position
        p[k+1] = rem_val;
        k++;
}
```